POLITECNICO DI TORINO

Master's Degree in Computer engineering

Master's Degree Thesis

Network connectivity and observability in multicluster environments

Supervisors

Prof. Fulvio RISSO

Dott. Marco IORIO

Dott. Aldo LACUKU

Candidate

Francesco CHEINASSO

Academic year 2021-2022

Summary

Nowadays cloud computing is becoming more important and the current trend is to engineer the new web applications to be cloud-native, to split up the application into micro-services, each one containerized and deployed as an independent part of a bigger application. A technology that broke through the cloud market is Kubernetes, a project that allows orchestrating containers in a cloud environment, creating an abstraction layer that hides the physical machines, allowing to manage them as a single entity. However, a new trend in the last years is to extend this abstraction to a group of clusters to create another abstraction layer to allow managing them like a single entity. This approach is called multi-clusters, a scenario where multiple independent clusters communicate and create a federation allowing to share resources between them and to exchange workloads. The advantage of a multi-clusters approach is the possibility to share the cluster's resources with others to optimize the usage of the latter. It also allows companies no longer depend on a specific cluster or cloud provider.

The goal of this thesis is to study how to interconnect more Kubernetes clusters and allows their microservices to communicate transparently. For this purpose two designs have been produced. These designs present a network model to enable communication with minimal dependencies and without the need for specific configurations on clusters. To test the new design in a real environment, the thesis design focuses on Liqo, an open-source project started at Politecnico di Torino. Each design has been implemented and validated. Finally, the two designs have been compared to evaluate which is the best. Another goal of the thesis is to provide the user with a simple way to monitor the status of the network and to keep track of the performance of the connectivity towards federated clusters. These observability features have been designed to expose information to other microservices.

Table of Contents

Chapter 1

Introduction

In the last several years, the ICT world has seen incredible innovation with the introduction of virtualization first, then with containerization, and finally with orchestrators. In this last field, one of the main actors is Kubernetes, an open-source system for managing containerized applications in a clustered environment. The spread of Kubernetes is rapidly increasing; in cloud providers such as Google Cloud Platform and Microsoft Azure it is the most popular choice [1] and many companies and organizations have started to set up their clusters in order to migrate their applications on it. With the advent of 5G and edge computing also telecommunications companies are moving towards Kubernetes-based solutions [2].

1.1 The need for Multiple Clusters

Organizations may need a multi-cluster environment for many different reasons. We can distinguish between two main categories of environments:

- In a **Cloud Environment**: where a single company can have many large data centers, both on-premise (in private infrastructure and on proprietary hardware) and on managed solutions (in a public cloud provider, like Amazon Web Services, Google Cloud Platform, Microsoft Azure, and many others).

  An organization may need multiple clusters in a cloud environment to have a high resource availability, distributed in multiple zones, and may require that these clusters are hosted by different cloud providers to contain costs or to not be strictly linked to a specific one of them.

  The usage of multiple clusters can also reduce scalability problems on very big clusters.

- In an **Edge or IoT Environment**: where a single company can have a lot of small clusters, even single-node ones. They can be geographically distributed to be closer to the end-user.

In this scenario, the main needs are the availability of the same API (for the software) and the re-utilization of the same skills (for the humans) already used in the cloud world to manage small devices. With the interoperability of the API, a new and closer integration becomes possible with the movement of the applications between different devices.

1.2 Multi-cluster and Liqo

Nowadays organizations don't need anymore more separated clusters, but they need these clusters to be able to communicate and cooperate. This is called **multi-clusters topology**, an approach where more clusters can behave like nodes of a bigger system. One of the main features needed is the possibility to share resources between them and exchange workloads dynamically and reactively.

Kubernetes does not support this approach, in fact, the maximum level of abstraction for Kubernetes is the **node** entity (see chapter 2), which is usually a single physical machine that is part of the cluster. The idea behind a multi-clusters approach in Kubernetes is to allow single clusters to use and share their nodes with another cluster. In this way is possible to have separated clusters from Kubernetes view and a logical **big cluster** (1.1)

Liqo is one of the solutions which allows enabling a multi-clusters approach inside Kubernetes and this thesis will use it as a workbench to study multi-clusters network designs.

1.3 Goal of the thesis

The goal of the thesis is to allow 2 clusters to communicate and inter-operate. In a multi-clusters environment, there are several challenges to face up. The one on which the thesis focuses is the exchange of network information. Network information is all the parameters that 2 clusters need to know to establish a connection. Data exchanged are usually details about the other cluster, that are needed to setup connections.

The first step to reach this goal is to study some possible network designs to understand what is the best in terms of simplicity, elasticity, and dynamism, reducing the amount of information that has to be shared and limiting race conditions and complexity.

Then another important step is how to integrate a strong and extensible mechanism to allow the observability of the network health status.

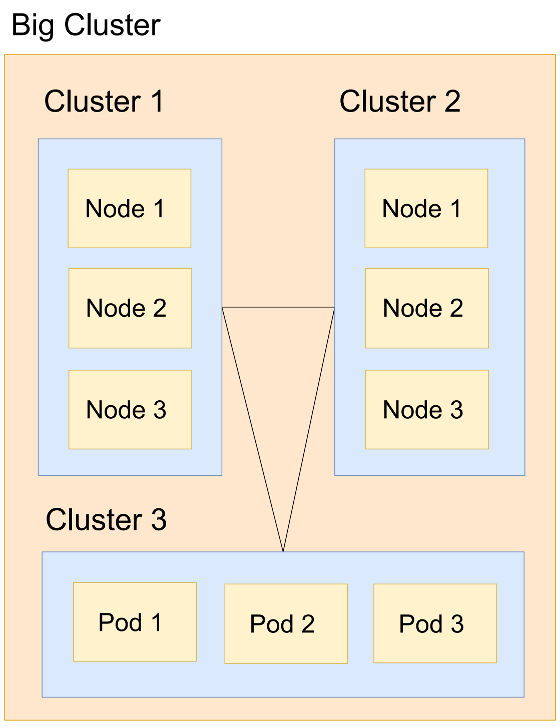


Figure 1.1: Multi-clusters topology

1.4 Structure of the work

This thesis is structured as follows:

- **Chapter 2** provides a presentation of Kubernetes, its architecture, and concepts;
- **Chapter 3** provides a presentation of Liqo, its architecture features, and concepts;
- **Chapter 4** provides a presentation of advanced network concepts used in the next chapters;
- **Chapter 5** analyzes possible cross-cluster network designs and their implementation with advantages and disadvantages;
- **Chapter 6** analyzes possible network observability designs and their implementation with advantages and disadvantages;
- **Chapter 7** evaluation of the obtained results, considering which is the best approach.

- **Chapter 8** conclusions about the thesis and future perspectives.

Chapter 2

Kubernetes

2.1 Kubernetes: a bit of history

Around 2004, Google created the **Borg** [3] system, a small project with less than 5 people initially working on it. The project was developed as a collaboration with a new version of Google's search engine. Borg was a large-scale internal cluster management system, which "ran hundreds of thousands of jobs, from many thousands of different applications, across many clusters, each with up to tens of thousands of machines" [3].

In 2013 Google announced **Omega** [4], a flexible and scalable scheduler for large compute clusters. Omega provided a "parallel scheduler architecture built around shared state, using lock-free optimistic concurrency control, in order to achieve both implementation extensibility and performance scalability".

In the middle of 2014, Google presented **Kubernetes** as on open-source version of Borg. Kubernetes was created by Joe Beda, Brendan Burns, and Craig McLuckie, and other engineers at Google. Its development and design were heavily influenced by Borg and many of its initial contributors previously used to work on it. The original Borg project was written in C++, whereas for Kubernetes the Go language was chosen.

In 2015 Kubernetes v1.0 was released. Along with the release, Google set up a partnership with the Linux Foundation to form the **Cloud Native Computing Foundation** (CNCF) [5]. Since then, Kubernetes has significantly grown, achieving the CNCF graduated status and being adopted by nearly every big company. Nowadays it has become the de-facto standard for container orchestration [6, 7].

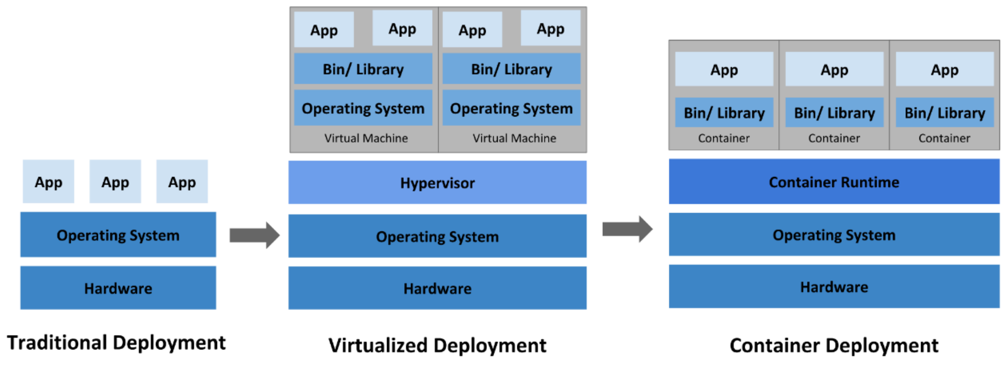


Figure 2.1: Evolution of applications deployments

2.2 Evolution of workloads management

Traditional deployment era In the traditional deployment era, organizations ran applications on physical servers. There was no way to define application constraints to limit resource usage, and some applications would end up taking most of the resources available, making the remaining applications starve. This led system managers to deploy one server per application, increasing costs and maintenance work. At this point, the community rediscovered the abandoned concept of virtualization.

Virtualized deployment era In the virtualized deployment era, developers could run multiple Virtual Machines (VMs) on a single physical server, and ensure applications would not interfere with one another, by running one VM per application. Virtualization allows defining resource-usage constraints for each VM, and makes software running on one VM isolated from the rest of the system and other VMs, leading to a much more stable and secure environment, as applications cannot interfere with one another, nor freely access private application data. Moreover, it allows better scalability as application instances can be scaled up or down easily by spawning or deleting VMs as needed. Each VM includes a whole operating system and can be tweaked to include the properly versioned dependencies as requested by the running application: this creates sealed compartments that are easy to manage and maintain, as well as to debug. Overall, less physical servers are deployed, costs are lower and companies can get the most out of their available servers, preventing them from being underused.

Containerized deployment era The next step in the evolution of workloads deployment came with the rise of containerization. Containers work similarly to VMs, but with less strict isolation properties so that different applications can share the same Operating System. For this, they are considered lightweight. Just like VMs, containers have their own filesystem, share of CPU, memory, process space, and more. Containers are decoupled from the underlying infrastructure: this makes them portable across clouds and OS distributions. What makes them so popular is the set of extra benefits they provide, such as:

- The agile application creation and deployment, given the ease of creation of container images compared to VM images.
- Continuous development, integration and deployment, thanks to the reliable and frequent container image build and deployments.
- Application health checks and observability.
- Cloud and OS distribution portability.
- Application-centric management, raising the abstraction level in order to simply focus on running the application.
- Resource utilization that yields high efficiency and density.

In parallel to the sheer technological advancements, an improvement on the workload management methods has been observed: from handling VMs as single entities, we moved to a "cattle" model where VMs were handled in a more general way (although their management would still be quite coupled to their lives), to move further and reach a decoupled approach, that is the one used by Kubernetes: a declarative way that expresses general intentions that are taken by the system and applied to all of the interested resources, without having to deal with the single instances, resulting in a more detached view where resources are seen as commodities that can be created, destroyed, and replaced as needed.

In this chapter we analyse Kubernetes architecture, showing also its history and evolution through time, in order to lay the foundations for all the work which will be exposed later on. Kubernetes (often shortened as K8s) is a huge framework and a deep examination of it would require much more time and discussion, hence we only provide here a description of its main concepts and components. Further details can be found in the official documentation [8].

2.3 Container orchestrators

When hundreds or thousands of containers are created, the need of a way to manage them becomes essential; container orchestrators serve this purpose. A container orchestrator is a system designed to easily manage complex containerization deployments across multiple machines from one central location. As depicted in figure 2.2, Kubernetes is by far the most used container orchestrator. We provide a description of such system in the following.

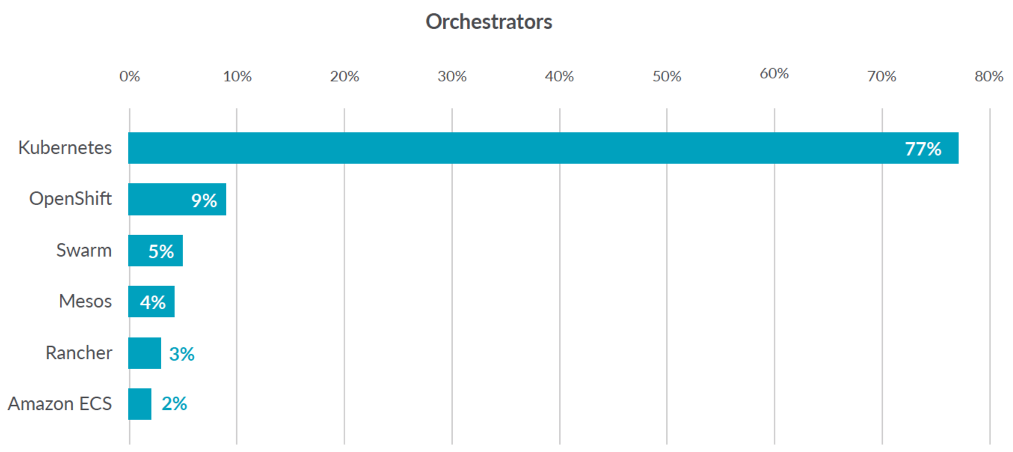


Figure 2.2: Container orchestrators use [9].

Kubernetes provides many services, including:

- **Service discovery and load balancing** A container can be exposed using the DNS name or using its own IP address. If traffic to a container is high, a load balancer able to distribute the network traffic is provided.
- **Storage orchestration** A storage system can be automatically mounted, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks** The desired state for the deployed containers can be described, and the actual state can be changed to the desired state at a controlled rate. For example, it is possible to automate the creation of new containers of a deployment, remove existing containers and adopt all their resources to the new container.
- **Automatic bin packing** Kubernetes is provided with a cluster of nodes that can be used to run containerized tasks. It is possible to set how much CPU and memory (RAM) each container needs, and automatically the containers are sized to fit in the nodes to make the best use of the resources.
- **Secret and configuration management** It is possible to store and manage sensitive information in Kubernetes, such as passwords, OAuth tokens, and SSH keys. It is possible to deploy and update secrets and application configuration without rebuilding the container images, and without exposing secrets in the stack configuration.

2.4 Kubernetes architecture

When Kubernetes is deployed, a cluster is created. A Kubernetes cluster consists of a set of machines, called **nodes**, that run containerized applications. At least one of the nodes hosts the control plane and is called **master**. Its role is to manage the cluster and expose an interface to the user. The **worker** node(s) host the **pods** that are the components of the application. The master manages the worker nodes and the pods in the cluster. In production environments, the control plane usually runs across multiple machines and a cluster runs on multiple nodes, providing fault-tolerance and high availability.

Figure 2.3 shows the diagram of a Kubernetes cluster with all the components linked together.

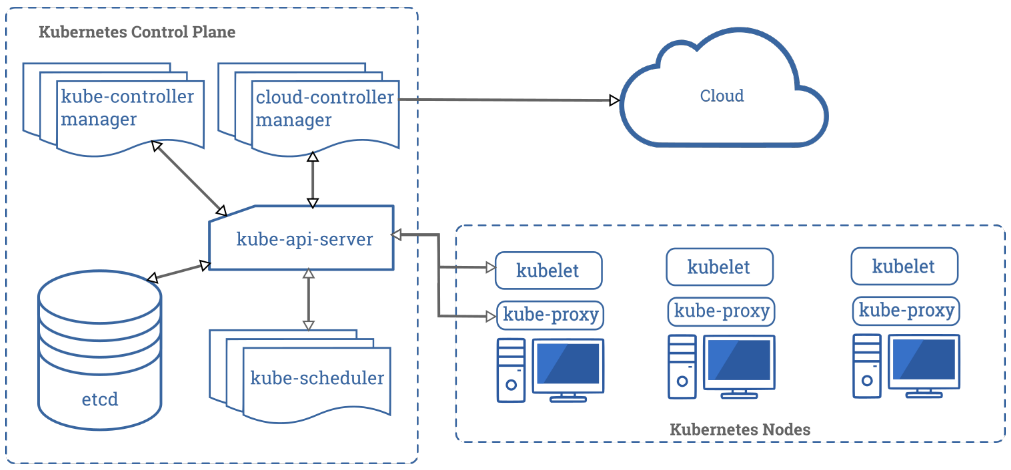


Figure 2.3: Kubernetes architecture

2.4.1 Control plane components

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod). Although they can be run on any machine in the cluster, for simplicity, they are typically executed all together on the same machine, which does not run user containers.

API server

The API server is the component of the Kubernetes control plane that exposes the Kubernetes REST API, and constitites the front end for the Kubernetes control plane. Its function is to intercept REST request, validate and process them. The main implementation of a Kubernetes API server is `kube-apiserver`. It is designed to scale horizontally, which means it scales by deploying more instances. Moreover, it can be easily redounded to run several instances of it and balance traffic among them.

etcd

`etcd` is a distributed, consistent and highly-available key value store used as Kubernetes' backing store for all cluster data. It is based on the Raft consensus algorithm, which allows different machines to work as a coherent group and survive to the breakdown of one of its members. `etcd` can be stacked in the master node or external, installed on dedicated host. Only the API server can communicate with it.

Scheduler

The scheduler is the control plane component responsible of assigning the pods to the nodes. The one provided by Kubernetes is called `kube-scheduler`, but it can be customized by adding new schedulers and indicating in the pods to use them. `kube-scheduler` watches for newly created pods not assigned to a node yet, and selects one for them to run on. To make its decisions, it considers singular and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines.

kube-controller-manager

Component that runs controller processes. It continuously compares the desired state of the cluster (given by the objects specifications) with the current one (read from `etcd`). Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process. These controllers include:

- Node Controller: responsible for noticing and reacting when nodes go down.
- Replication Controller: in charge of maintaining the correct number of pods for every replica object in the system.
- Endpoints Controller: populates the Endpoint objects (which links Services and Pods).
- Service Account & Token Controllers: create default accounts and API access tokens for new namespaces.

cloud-controller-manager

This component runs controllers that interact with the underlying cloud providers. The `cloud-controller-manager` binary is a beta feature introduced in Kubernetes 1.6. It only runs cloud-provider-specific controller loops. You can disable these controller loops in the `kube-controller-manager`.

`cloud-controller-manager` allows the cloud vendor's code and the Kubernetes code to evolve independently of each other. In prior releases, the core Kubernetes code was dependent upon cloud-provider-specific code for functionality. In future releases, code specific to cloud vendors should be maintained by the cloud vendor themselves, and linked to `cloud-controller-manager` while running Kubernetes. Some examples of controllers with cloud provider dependencies are:

- Node Controller: checks the cloud provider to update or delete Kubernetes nodes using cloud APIs.
- Route Controller: responsible for setting up network routes in the cloud infrastructure.
- Service Controller: for creating, updating and deleting cloud provider load balancers.
- Volume Controller: creates, attaches, and mounts volumes, interacting with the cloud provider to orchestrate them.

2.4.2 Node components

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

Container Runtime

The `container runtime` is the software that is responsible for running containers. Kubernetes supports several container runtimes: Docker, containerd, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface).

kubelet

An agent that runs on each node in the cluster, making sure that containers are running in a pod. The `kubelet` receives from the API server the specifications of the Pods and interacts with the `container runtime` to run them, monitoring their state and assuring that the containers are running and healthy. The connection with the `container runtime` is established through the Container Runtime Interface and is based on gRPC.

kube-proxy

`kube-proxy` is a network agent that runs on each node in your cluster, implementing part of the Kubernetes Service concept. It maintains network rules on nodes, which allow network communication to your Pods from inside or outside of the cluster. If the operating system is providing a packet filtering layer, `kube-proxy` uses it, otherwise it forwards the traffic itself.

Addons

Features and functionalities not yet available natively in Kubernetes, but implemented by third parties pods. Some examples are DNS, dashboard (a web gui), monitoring and logging.

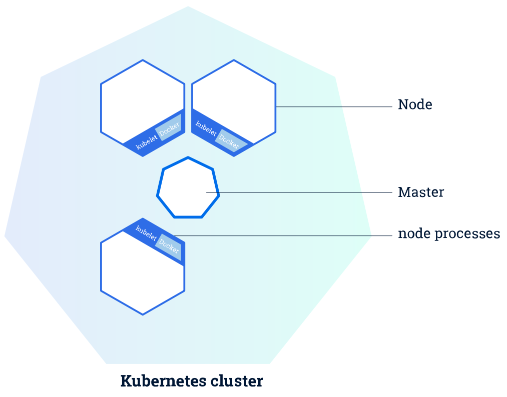


Figure 2.4: Kubernetes master and worker nodes [8].

2.5 Kubernetes objects

Kubernetes defines several types of objects, which constitutes its building blocks. Usually, a K8s resource object contains the following fields:

- `apiVersion`: the versioned schema of this representation of the object;
- `kind`: a string value representing the REST resource this object represents;
- `ObjectMeta`: metadata about the object, such as its name, annotations, labels etc.;
- `ResourceSpec`: defined by the user, it describes the desired state of the object;

- `ResourceStatus`: filled in by the server, it reports the current state of the resource.

The allowed operations on these resources are the typical CRUD actions:

- **Create**: create the resource in the storage backend; once a resource is created, the system applies the desired state.
- **Read**: comes with 3 variants
  - **Get**: retrieve a specific resource object by name;
  - **List**: retrieve all resource objects of a specific type within a namespace, and the results can be restricted to resources matching a selector query;
  - **Watch**: stream results for an object(s) as it is updated.
- **Update**: comes with 2 forms
  - **Replace**: replace the existing spec with the provided one;
  - **Patch**: apply a change to a specific field.
- **Delete**: delete a resource; depending on the specific resource, child objects may or may not be garbage collected by the server.

In the following we illustrate the main objects needed in the next chapters.

2.5.1 Label & Selector

Labels are key-value pairs attached to a K8s object and used to organize and mark a subset of objects. Selectors are the grouping primitives which allow to select a set of objects with the same label.

2.5.2 Namespace

Namespaces are virtual partitions of the cluster. By default, Kubernetes creates 4 Namespaces:

- **kube-system**: it contains objects created by K8s system, mainly control-plane agents;
- **default**: it contains objects and resources created by users and it is the one used by default;
- **kube-public**: readable by everyone (even not authenticated users), it is used for special purposes like exposing cluster public information;

- **kube-node-lease**: it maintains objects for heartbeat data from nodes.

It is a good practice to split the cluster into many Namespaces in order to better virtualize the cluster.

2.5.3 Pod

Pods are the basic processing units in Kubernetes. A pod is a logic collection of one or more containers which share the same network and storage, and are scheduled together on the same pod. Pods are ephemeral and have no auto-repair capacities: for this reason they are usually managed by a controller which handles replication, fault-tolerance, self-healing etc.

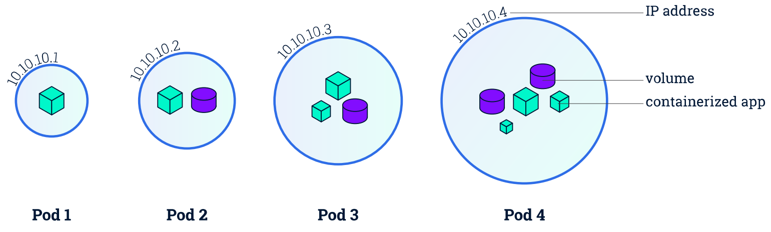


Figure 2.5: Kubernetes pods [8].

2.5.4 ReplicaSet

ReplicaSets control a set of pods allowing to scale the number of pods currently in execution. If a pod in the set is deleted, the ReplicaSet notices that the current number of replicas (read from the `Status`) is different from the desired one (specified in the `Spec`) and creates a new pod. Usually ReplicaSets are not used directly: a higher-level concept is provided by Kubernetes, called **Deployment**.

2.5.5 Deployment

Deployments manage the creation, update and deletion of pods. A Deployment automatically creates a ReplicaSet, which then creates the desired number of pods. For this reason an application is typically executed within a Deployment and not in a single pod. The listing 2.1 is an example of deployment.

Listing 2.1: Basic example of Kubernetes Deployment [8].

```
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

The code above allows to create a Deployment with name `nginx-deployment` and a label `app`, with value `nginx`. It creates three replicated pods and, as defined in the `selector` field, manages all the pods labelled as `app:nginx`. The template field shows the information of the created pods: they are labelled `app:nginx` and launch one container which runs the nginx DockerHub image at version 1.7.9 on port 80.

2.5.6 DaemonSet

A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created [8]. Some typical uses of a DaemonSet are:

- running a cluster storage daemon on every node;
- running a logs collection daemon on every node;
- running a node monitoring daemon on every node.

Listing 2.2: Basic example of Kubernetes daemonset [8].

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
```

2.5.7 Service

A Service is an abstract way to expose an application running on a set of Pods as a network service. It can have different access scopes depending on its ServiceType:

- **ClusterIP**: Service accessible only from within the cluster, it is the default type;
- **NodePort**: exposes the Service on a static port of each Node's IP; the NodePort Service can be accessed, from outside the cluster, by contacting `<NodeIP>:<NodePort>`;
- **LoadBalancer**: exposes the Service externally using a cloud provider's load balancer;
- **ExternalName**: maps the Service to an external one so that local apps can access it.

The following Service is named `my-service` and redirects requests coming from TCP port 80 to port 9376 of any Pod with the `app=MyApp` label.

Listing 2.3: Basic example of Kubernetes Service [8].

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
```

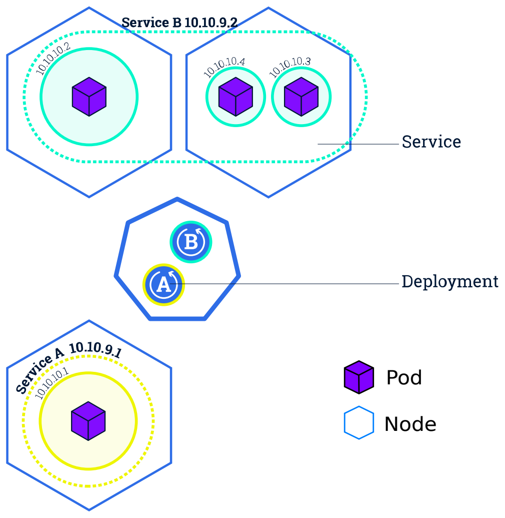


Figure 2.6: Kubernetes Services [8].

```
  selector:
    app: myApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

2.6 Kubernetes network architecture

Kubernetes defines a network model that helps provide simplicity and consistency across a range of networking environments and network implementations. The Kubernetes network model provides the foundation for understanding how containers, pods, and services within Kubernetes communicate with each other [10]. The Kubernetes network model specifies:

1. Every pod gets its own IP address;
2. Containers within a pod share the pod IP address and can communicate freely with each other;
3. Pods can communicate with all other pods in the cluster using pod IP addresses (without NAT);

4. Agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node;

5. Pods in the host network of a node can communicate with all pods on all nodes (without NAT);

6. Isolation (restricting what each pod can communicate with) is defined using network policies.

As a result, pods can be treated much like VMs or hosts (they all have unique IP addresses), and the containers within pods very much like processes running within a VM or host (they run in the same network namespace and share an IP address). This model makes it easier for applications to be migrated from VMs and hosts to pods managed by Kubernetes. In addition, because isolation is defined using network policies rather than the structure of the network, the network remains simple to understand. This style of network is sometimes referred to as a “flat network”.

2.6.1 Container communication within same pod

Containers in a Pod are accessible via `localhost`, they use the same network namespace. For containers, the observable host name is a Pod’s name. Since containers share the same IP address and port space, different ports in containers for incoming connections must be used. Because of this, applications in a Pod must coordinate their usage of ports.

2.6.2 Pod communication within the same node

Before the infrastructure container is started, a virtual Ethernet interface pair (a veth pair) is created for the container. One interface of the veth pair stays in the host’s namespace (it tagged with vethxxx) while the other interface is moved into the container’s network namespace and renamed to eth0. These two virtual interfaces are like two ends of a pipe that everything goes in one side, comes out on the other.The interface in the host’s network namespace is attached to a network bridge that container runtime is configured to use. The eth0 interface in the container is assigned an IP address from the bridge’s address range. Anything that application running inside the container sends to the eth0 network interface and comes out at the other veth Interface in host’s namespace and is sent to bridge. So, any network connected to the bridge can receive it.

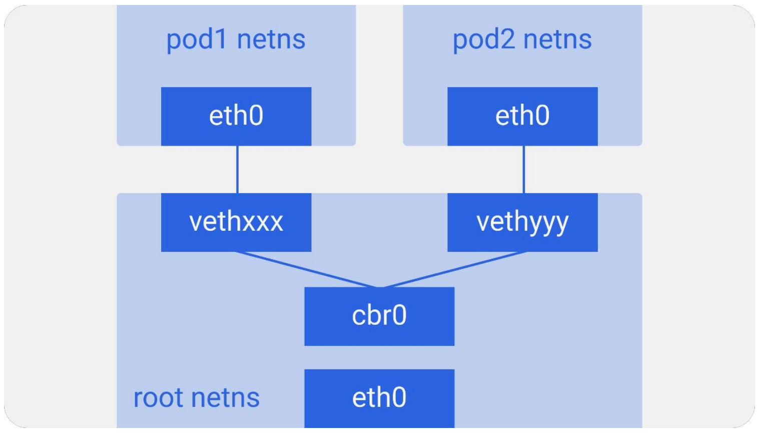


Figure 2.7: Pod to pod communication within same node.

2.6.3 Pod communication on different nodes

Pod IP addresses must be unique across the whole cluster, so the bridges across the nodes must use non-overlapping address ranges to prevent pods from different nodes from getting the same IP address. There are many methods for connecting the bridges on different nodes. This can be done with overlay or underlay networks or by regular layer 3 routing(direct routing).

2.6.4 CNI (Container Network Interface)

CNI (Container Network Interface) is a Cloud Native Computing Foundation project consisting of a specification and libraries for writing plugins to configure network interfaces in Linux containers. CNI concerns itself only with network connectivity of containers and removing allocated resources when the container is deleted. Kubernetes uses the CNI specifications and plug-ins to orchestrate networking. Also, it can address other container's IP addresses without using the Network Address Translation (NAT). Every time a Pod is initialized or removed, the default CNI plug- in is called with the default configuration, which this CNI plug-in creates a pseudo interface, attaches it to the underlay network, sets IP Address, routes, and maps it to the Pod namespace. It should be passed –network-plugin = cni to the Kubelete when launching it for using the CNI plugin. If the environment is not using the default configuration directory (`/etc/cni.net.d`), the CNI plugin passes the correct configuration directory as a value to `-cni-conf-dir`. Moreover, the Kubelet looks for the CNI plugin binary at `/opt/cni/bin`, but it

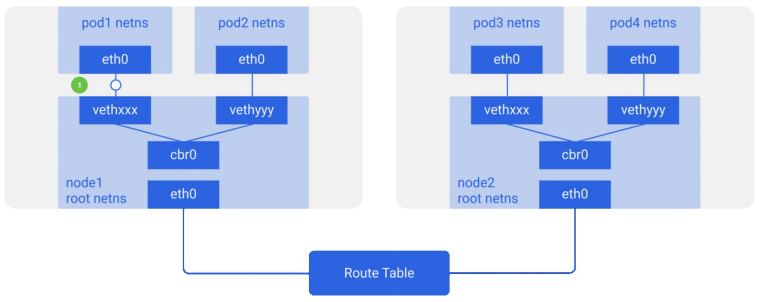


Figure 2.8: Pod to pod communication across different nodes.

can be specified an alternative location with `-cni-bin-dir`.

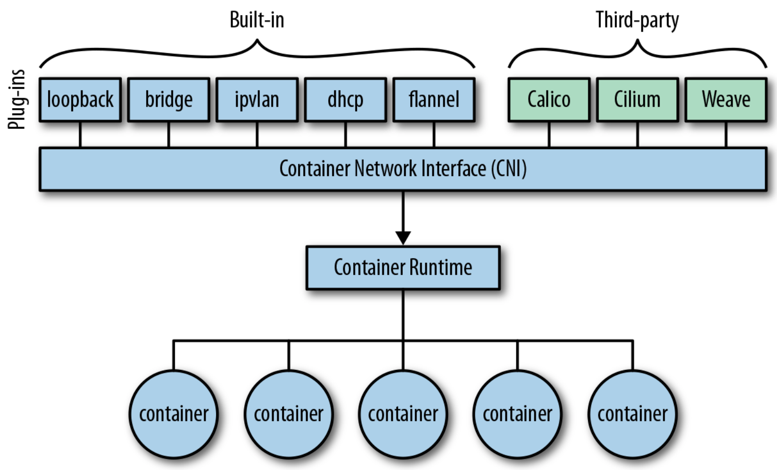


Figure 2.9: Container network interface [11]

2.6.5 Pod to service networking

Pod IP addresses are not durable and will appear and disappear in response to scaling up or down, application crashes, or node reboots. Each of these events can make the Pod IP address change without warning. Services were built into Kubernetes to address this problem. The Kubernetes service manages the state of

Pods, allowing us to track a set of the pod IP address that dynamically changes over time. Services act as an abstraction over Pods and assign a single virtual IP address to a group of Pod IP addresses. Any traffic addressed to the virtual IP of the service will be routed to the set of Pods that are associated with the virtual IP. This allows the set of Pods associated with a service to change at any time clients only need to know the service's virtual IP, which does not change [12].

2.7 Kubebuilder

Kubebuilder is a framework for building Kubernetes APIs using Custom Resource Definitions (CRDs) [13].

CustomResourceDefinition is an API resource offered by Kubernetes which allows to define Custom Resources (CRs) with a name and schema specified by the user. When a new CustomResourceDefinition is created, the Kubernetes API server creates a new RESTful resource path; the CRD can be either namespaced or cluster-scoped. The name of a CRD object must be a valid DNS subdomain name.

A **Custom Resource** is an endpoint in the Kubernetes API that is not available in a default Kubernetes installation and which frees users from writing their own API server to handle them [8]. On their own, custom resources simply let you store and retrieve structured data. In order to have a more powerful management, you also need to provide a custom controller which executes a control loop over the custom resource it watches: this behaviour is called Operator pattern [14].

Kubebuilder helps a developer in defining his Custom Resource, taking automatically basic decisions and writing a lot of boilerplate code. These are the main actions operated by Kubebuilder [13]:

1. Create a new project directory.
2. Create one or more resource APIs as CRDs and then add fields to the resources.
3. Implement reconcile loops in controllers and watch additional resources.
4. Test by running against a cluster (self-installs CRDs and starts controllers automatically).
5. Update bootstrapped integration tests to test new fields and business logic.
6. Build and publish a container from the provided Dockerfile.

Chapter 3

Liqo

This chapter introduces the conceptual foundations at the base of Liqo [15], as well as the core elements that make up its architecture.

3.1 Liqo: an overview

The Kubernetes technology is widely employed to handle cloud tasks. Clusters are designed to provide more resources—in terms of sheer computing power, available memory, storage capacity—than the ones normally required, to handle temporary peaks of load. This means that this excess of capabilities could be used by other clusters that undergo a period of higher load. Liqo [16] aims to unleash this potential power by connecting clusters together and have them work synergically to pursue their goals.

To accomplish this task, clusters establish a peering session that results in a larger virtual cluster that hosts the sum of the resources exposed by each cluster involved in the peering process.

The benefit of using Liqo is that it takes the core concepts that are well-known in the Kubernetes environment and exploits them to achieve more possibilities. Indeed, a cluster sees its peers simply as (virtual) nodes that add up to its (physical) ones, and schedules tasks to its nodes regardless of their actual nature.

The next sections will describe more in depth the presented concepts, starting with a core element and how to establish it: the Liqo peering.

3.2 The Liqo peering

Once two (or more) Kubernetes clusters are available to host workloads, they can become part of a multi-cluster topology by activating a peering session between them. This is where the Liqo experience starts off. A Liqo peering takes separate entities and joins them into a wider environment that is capable of handling larger workloads. As a result, each involved cluster becomes aware of the existence of other remote peers, modeled by the ForeignCluster Custom Resource (CR). This process entails the exchange network parameters and other cluster information, so as to create a secure VPN that pods will leverage to communicate with one another as part of a large distributed cross-cluster application.

Cluster peerings are not required to be symmetric. Their flexibility allows a cluster to establish:

- **An outgoing peering**, so that the cluster can offload its workloads, but won't receive any by its peer.
- **An incoming peering**, so that the cluster hosts remote workloads, but won't offload any to its peer.
- **A bidirectional peering**, the union of the two above.

When an outgoing peering is active, it is of paramount importance to control what could be offloaded and what should not. This is done by leveraging some native Kubernetes concepts, namely Namespaces and label selectors, and some logic provided by Liqo to select which namespaces to offload, which pods within such namespaces to offload, and even which remote peers as the target of this offloading mechanism. The possibilities are endless.

The basic requirements to start a peering session is to have access to the remote Kubernetes API Server. This allows clusters to exchange information and create resources remotely, with the result of having a VPN that remote pods use to communicate as if they were all in the same Kubernetes cluster.

3.3 The Liqo reflection

Once a peering is established, the workload offloading is enabled by leveraging the virtual node abstraction and the namespace extension.

A virtual node represents a remote cluster and all of its shared resources (e.g. CPU and memory). This allows for a transparent extension of the local cluster's resources, as the virtual node added to the cluster is seamlessly taken into account by the vanilla Kubernetes scheduler when selecting the best place for executing workloads.

In addition to that, Liqo enables the extension of Kubernetes namespaces across the cluster boundaries. Once a namespace is selected for offloading, Liqo automatically creates twin namespaces in the selected subset of remote peers. These remote twin namespaces will host the remotely offloaded pods, as well as other resources living in the local namespace that has been extended remotely, such as those related to service exposition (Ingress, Service and Endpoints resources), or storing configuration data (ConfigMaps and Secrets), to name a few.

3.4 Network Fabric

The network fabric is the Liqo subsystem transparently extending the Kubernetes network model across multiple independent clusters, such that offloaded pods can communicate with each other as if they were all executed locally.

In detail, the network fabric ensures that all pods in a given cluster can communicate with all pods on all remote peered clusters, either with or without NAT translation. The support for arbitrary clusters, with different parameters and components (e.g., CNI plugins), makes it impossible to guarantee non-overlapping pod IP address ranges (i.e., PodCIDR). Hence, possibly requiring address translation mechanisms, provided that NAT-less communication is preferred whenever address ranges are disjointed.

The figure 3.1 represents at a high level the network fabric established between two clusters, with its main components detailed in the following.

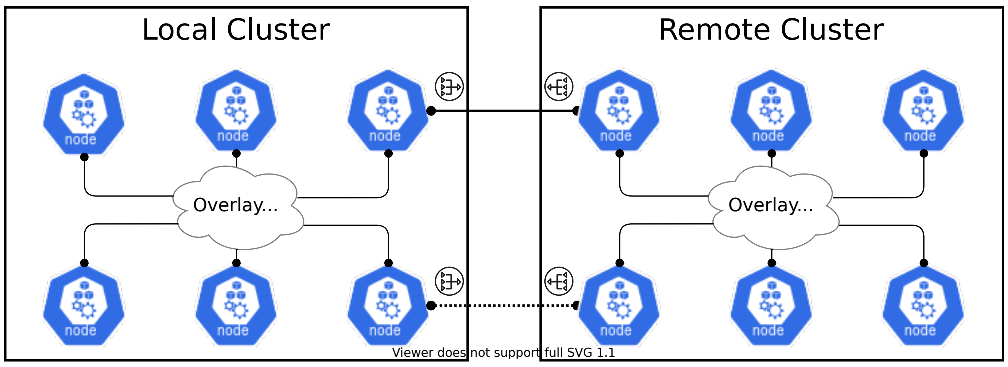


Figure 3.1: Network Fabric

3.4.1 Cross-cluster VPN tunnels

The interconnection between peered clusters is implemented through secure VPN tunnels, made with WireGuard, which are dynamically established at the end of the peering process, based on the negotiated parameters.

Tunnels are set up by the Liqo gateway, a component of the network fabric that is executed as a privileged pod on one of the cluster nodes. Additionally, it appropriately populates the routing table, and configures, by leveraging iptables, the NAT rules requested to comply with address conflicts.

Although this component is executed in the host network, it relies on a separate network namespace and policy routing to ensure isolation and prevent conflicts with the existing Kubernetes CNI plugin. Moreover, active/standby high-availability is supported, to ensure minimum downtime in case the main replica is restarted.

3.4.2 In-cluster overlay network

The overlay network is leveraged to forward all traffic originating from local pods/nodes, and directed to a remote cluster, to the gateway, where it will enter the VPN tunnel. The same process occurs on the other side, with the traffic that exits from the VPN tunnel entering the overlay network to reach the node hosting the destination pod.

Liqo leverages a VXLAN-based setup, which is configured by a network fabric component executed on all physical nodes of the cluster (i.e., as a DaemonSet). Additionally, it is also responsible for the population of the appropriate routing entries to ensure correct traffic forwarding.

3.5 Liqo Custom Resources

The following subsections present some of the Custom Resources used by Liqo to provide the peering and reflection features.

3.5.1 The NetworkConfig CR

This CR represents a set of network parameters (mainly IP addresses) by means of which clusters know how a remote peer has remapped the local PodCIDR, as well as the remote peer's PodCIDR. The "spec" part includes data related to the local cluster, while its "status" part reports the changes to the specifications. The idea is that a cluster creates this CR and sends it to the remote cluster it is going to establish a peering with. The remote cluster processes this CR and annotates in the "status" part everything it had to change in terms of IP address ranges to avoid any conflicts. These updates are reported back to the owning cluster.

Concurrently, the same happens in the opposite direction, so the remote cluster generates a NetworkConfig, writes its "spec" part and sends it to the local cluster, which annotates any changes in the "status" part to make the remote cluster aware of any modifications to the original specifications.

Once both the CRs are processed, a Liqo control loop reconciles them to create the TunnelEndpoint CR.

3.5.2 The TunnelEndpoint CR

This CR contains the relevant network configuration to establish a VPN tunnel with the remote cluster. This is used to make pods reach out to other remote pods as if they were in the same network.

3.5.3 The ForeignCluster CR

This CR models a remote cluster. It contains the details about the peering session that is in place between two clusters, such as whether the peering has been established successfully and what direction it takes (outgoing, incoming, or both). A ForeignCluster is created starting from the NetworkConfigs that the two parties have exchanged and processed.

3.5.4 The ShadowPod CR

When a Pod is scheduled onto a virtual node, a Pod is created in the remote cluster for the actual workload execution. In the remote cluster, a new object paired with the remote Pod is created: this is the ShadowPod. This resource, combined with its controller, guarantees the presence of the pod in the remote cluster, also in cases of connection faults.

3.6 Liqo Components

3.6.1 The CRD Replicator component

This component is dedicated to the reflection of some Liqo CRs just presented. To do so, it requires access to the remote API Server. It is a core element as it implements the network parameter exchange between clusters to set up the TunnelEndpoint CRs which will later be used respectively to keep track of the active peering sessions and to ensure remote pod-to-pod communications. The replicated CRDs are:

- NetworkConfig
- ResourceRequest
- ResourceOffer
- NamespaceMapping

The CRD Replicator architecture is quite complex, but essentially it is implemented through a so-called reflector, which is a data structure containing the required objects and data to detect changes in local and remote namespaces (using local and remote informers), as well as to perform the traditional CRUD[1] operations in those namespaces (using local and remote clients). In particular, when an object, such as a NetworkConfig, is created in a namespace enabled for reflection and with the proper metadata labels set up, the local reflector (that is the one belonging to the cluster that created the object) follows these steps:

- It detects a new object to be reflected.
- It creates a copy of that object in the remote namespace by using a pre-configured client to access the remote API server.
- It listens to any changes occurring in the reflected object, which usually boils down to a status update performed by the remote cluster controllers, as happens with NetworkConfigs to let the sender cluster know about possible remappings.
- It listens to any changes occurring in the local original copy, such as a deletion that needs to propagate to the remote cluster's namespace so that the remote copy gets deleted as well.

3.6.2 The Virtual Kubelet component

This component is a custom version of the Virtual Kubelet project [17]. Whenever a peering session is established with a remote cluster, a dedicated instance of this component is created. Once created, it is used to offload pods to remote clusters, seen by the Kubernetes control plane as normal cluster nodes onto which to schedule a normal task. In addition to that, it is used to reflect core Kubernetes resources, such as Services and Endpoints: once deployed in a Liqo-enabled namespace, that is a namespace extended remotely, they will always be reflected to the selected remote peers.

3.6.3 The IPAM component

This component contains the logic that translates IP addresses back and forth and keeps track of all the possible remappings between the local cluster and the remote peers. It is fundamental within Liqo as it knows all the NAT rules that are used to avoid address conflicts.

[1]Create, read, update, and delete (CRUD) are the four basic operations of persistent storage.

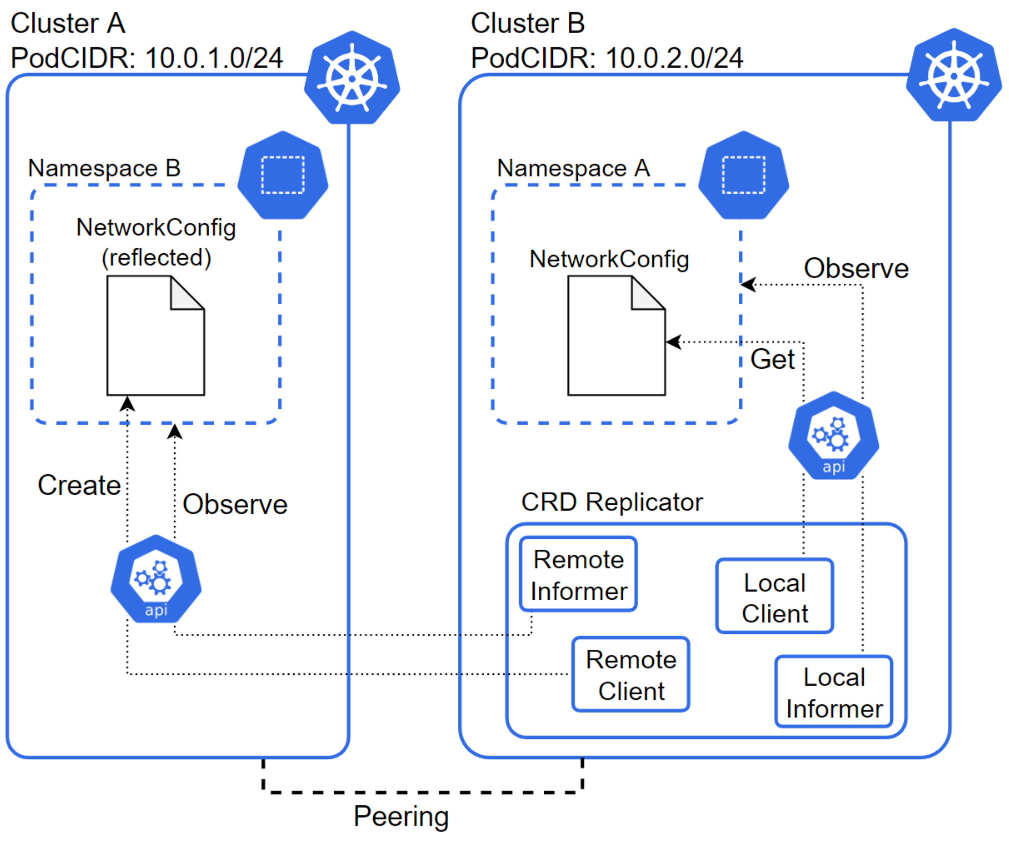


Figure 3.2: CRD Replicator

3.6.4 Network manager

The network manager (not shown in figure) represents the control plane of the Liqo network fabric. It is executed as a pod, and it is responsible for the negotiation of the connection parameters with each remote cluster during the peering process.

It features an IP Address Management (IPAM) plugin, which deals with possible network conflicts through the definition of high-level NAT rules (enforced by the data plane components). Additionally, it exposes an interface consumed by the reflection logic to handle IP addresses remapping. Specifically, this is leveraged to handle the translation of pod IPs (i.e., during the synchronization process from the remote to the local cluster), as well as during EndpointSlices reflection (i.e., propagated from the local to the remote cluster).

3.6.5 The Liqo Gateway

This component is responsible to manage connections with other clusters. All the traffic between two peered clusters has to pass through this component. It is possible to have more than just one Liqo Gateway, but only one at time can be activated and the others can be used in case of failures. The connection between clusters is managed with VPN tunnels and this component is the responsible for the management of them. Liqo support more VPN drivers (eg. Wireguard, OpenVPN, IPSec), providing an interface to implement the logic. However at the moment the only implemented driver is **Wireguard** (see section 4.3).

Chapter 4

Advanced Networking Concepts

In this chapter are explained some networking concepts and technologies have been used in the thesis implementation phase.

4.1 Linux Namespaces

Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources. The feature works by having the same namespace for a set of resources and processes, but those namespaces refer to distinct resources. Resources may exist in multiple spaces. Examples of such resources are process IDs, hostnames, user IDs, file names, and some names associated with network access, and interprocess communication. Namespaces are a fundamental aspect of containers on Linux.

4.1.1 Namespace kinds

Since kernel version 5.6, there are 8 kinds of namespaces. Namespace functionality is the same across all kinds: each process is associated with a namespace and can only see or use the resources associated with that namespace, and descendant namespaces where applicable. This way each process (or process group thereof) can have a unique view of the resources. Which resource is isolated depends on the kind of namespace that has been created for a given process group. Namespace kinds are:

- **Mount namespace**: controls mount points.

- **PID namespace**: provides processes with an independent set of process IDs.
- **Network namespace**: allows Linux network stack to behave in isolated groups.
- **IPC namespace**:allows processes to have separated IPC.
- **UTS namespace**: allows a single system to appear to have different host and domain names for different processes.
- **User namespace**: related to user privileges.
- **Control group namespace**: hides the identity of the cgroup of which process is a member.
- **Time namespace**: allows processes to see different system times.

4.2 Linux Network Stack

4.2.1 NetFiltert

The Netfilter framework within the Linux kernel is the basic building block on which packet selection systems like Iptables or the newer Nftables are built upon. It provides a bunch of hooks inside the Linux kernel, which are being traversed by network packets as those flow through the kernel (see figure 4.1). Other kernel components can register callback functions with those hooks, which enables them to examine the packets and make decisions on whether packets shall be dropped, accepted, or modified.

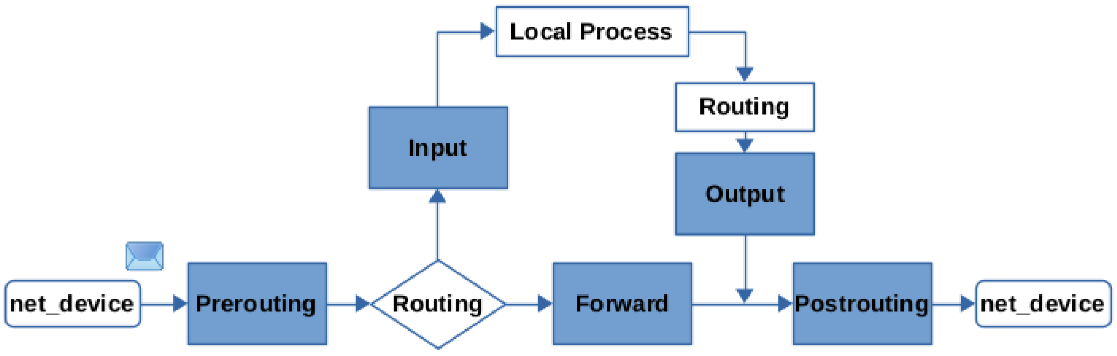


Figure 4.1: Netfilter stack overview

A network packet received on a network device first traverses the Prerouting hook. Then the routing decision happens and thereby the kernel determines whether this packet is destined at a local process (e.g. socket of a server listening on the system) or whether the packet shall be forwarded (in that case the system works as a router). In the first case, the packet then traverses the Input hook and is then given to the local process. In the second case, the packet traverses the Forward hook and finally the Postrouting hook, before being sent out on a network device. A packet that has been generated by a local process (e.g. a client or server software that likes to send something out on the network), first traverses the Output hook and then also the Postrouting hook, before it is sent out on a network device.

4.2.2 Iptables

iptables is a user-space utility program that allows a system administrator to configure the IP packet filter rules of the Linux kernel firewall. Iptables organizes its rules into tables and chains, whereas tables for the most part merely are a means to group chains together, which have something in common. E.g. chains that are used for nat belong to the nat table. The actual rules reside inside the chains. Iptables registers its chains with the Netfilter hooks by registering its hook functions as described above. This means when a network packet traverses a hook (e.g. Prerouting), then this packet traverses the chains which are registered with this hook and thereby traverses their rules.

In the case of Iptables all that is already pre-defined. A fixed set of tables exists, each table containing a fixed set of chains. The chains are named like the Netfilter hooks with which they are registered.

Table	**Contains chains**
filter	INPUT,FORWARD,OUTPUT
nat	PREROUTING, (INPUT), OUTPUT, POSTROUTING
mangle	PREROUTING, INPUT, FORWARD, OUTPUT, POSTROUTING
raw	PREROUTING, OUTPUT

The sequence in which the chains are being traversed when a packet traverses the hook (their priority) is also already fixed. The Netfilter packet flow image (4.1) shows this sequence in detail.

4.3 VPN - Wireguard

Wireguard is a communication protocol and free and open-source software that implements encrypted virtual private networks (VPN). In March 2020, the Linux version of the software reached a stable production release and was incorporated into the Linux 5.6 kernel. Wireguard is extremely simple yet fast and utilizes state-of-the-art cryptography. It aims to be faster, simpler, leaner, and more useful than IPsec. It intends to be considerably more performant than OpenVPN.

To create a wireguard connection between two hosts the first step is to create a wireguard network interface on each host. This interface has public and private keys, used to encrypt the communication between the hosts. An important detail about wireguard interfaces, which will be fundamental in the next chapter, is that a wireguard interface can support more **peers**. So if a host has to connect to other two different hosts, there are two possible solutions. The first is to create a dedicated interface for each remote host, the second is to use just one interface and create two different peers for the remote hosts inside the same interface.

Liqo and liqo-gateway have been designed to support more VPN drivers. At the moment the only implementation available is for the wireguard's driver. It is important to clarify this aspect because (as will be explained in chapter 6) parts of the thesis focused on the implementation of new features inside the VPN drivers. That's why this explanation about how wireguard works is necessary, even if Liqo supports different VPN drivers (see section 3.6.5).

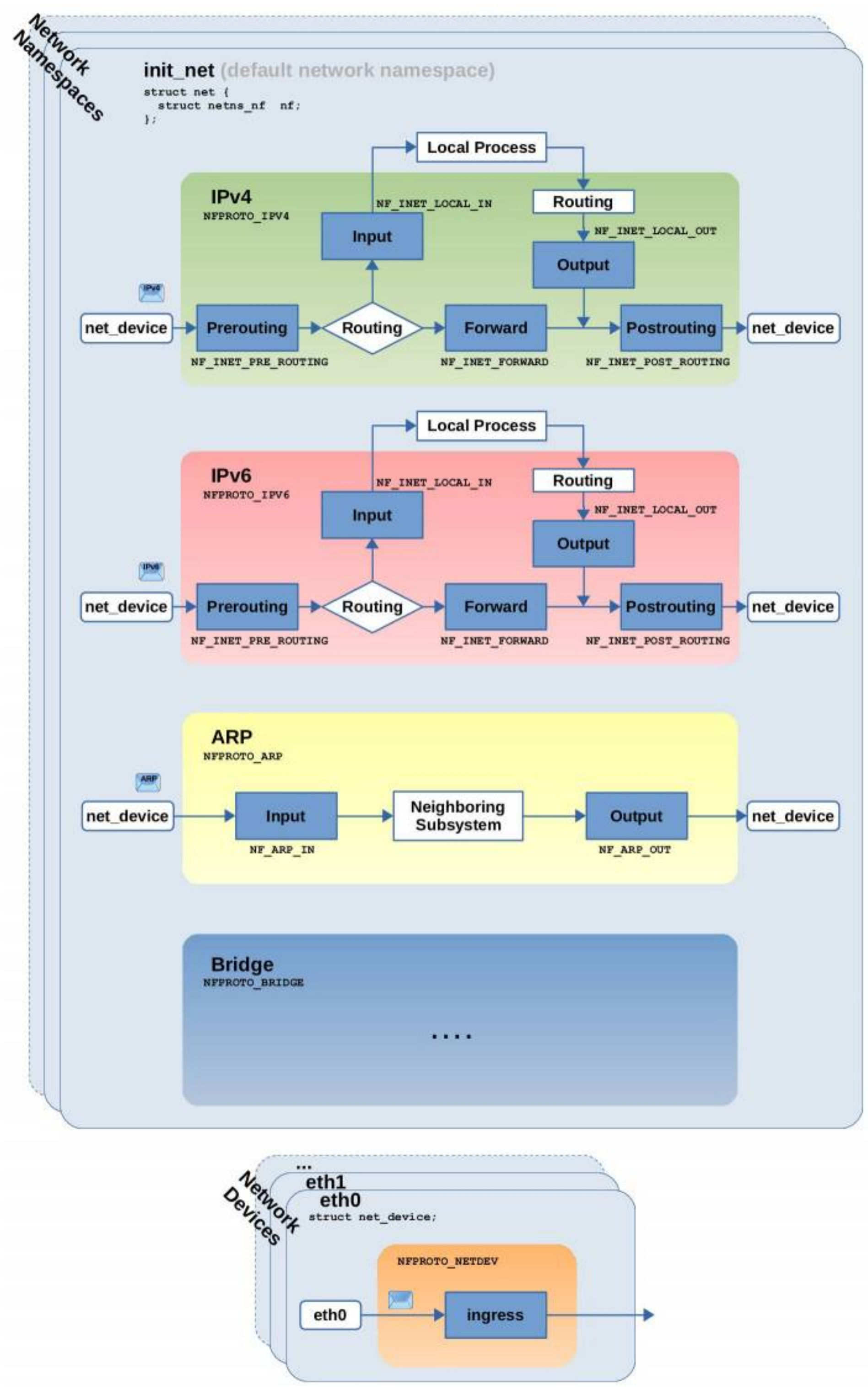


Figure 4.2: Netfilter stack details overview

Chapter 5

Cross-Cluster Network Design

Possible network designs to allows inter-cluster communication has been studied and evaluated. This chapter presents in detail the two main architectures designs proposed and their implementations.

5.1 The Problem

When two clusters are peered they need some unique IPs to allow pods to communicate. However, two clusters could have the same PodCIDR and the used IPs can be replicated. This problem could be solved by setting up a different PodCIDR for each cluster. This can work but introduces a strong constraint that would be better to avoid. Another possible solution is to rely on a **remapping mechanism**.

5.1.1 Remapping

The idea of remapping is to have a component, between two clusters, that converts a repeated IP to a valid one. For example, if two clusters have the same PodCIDR (eg. 10.0.0.0/16) could be a problem. Indeed if a pod with IP 10.0.0.5 is present in both clusters communication between the two pods will be impossible, because the two entities have the same address. To solve this problem is necessary to associate another IP with the original one. For example in figure 5.1 **cluster1** is able to send a packet to **cluster2**'s pod, even if **Pod1** and **Pod2** have the same IP. That's possible because **cluster1** autonomously associated another IP to **Pod2**, which is **20.0.0.5**. **Cluster 1** is able to send packets to **20.0.0.5** and this packet will be translated into a correct one, which can be received by **cluster2**. This process is called **Remapping**.

So if the **Pod1** inside the **Cluster1** needs to communicate with **Pod2** in **Cluster2**, **Cluster1** can identify all the pods inside **Cluster2** with a remapped IP. When the IP packets will be sent to **Pod2** the destination and source IP will be translated with the corresponding **remapped IP**.

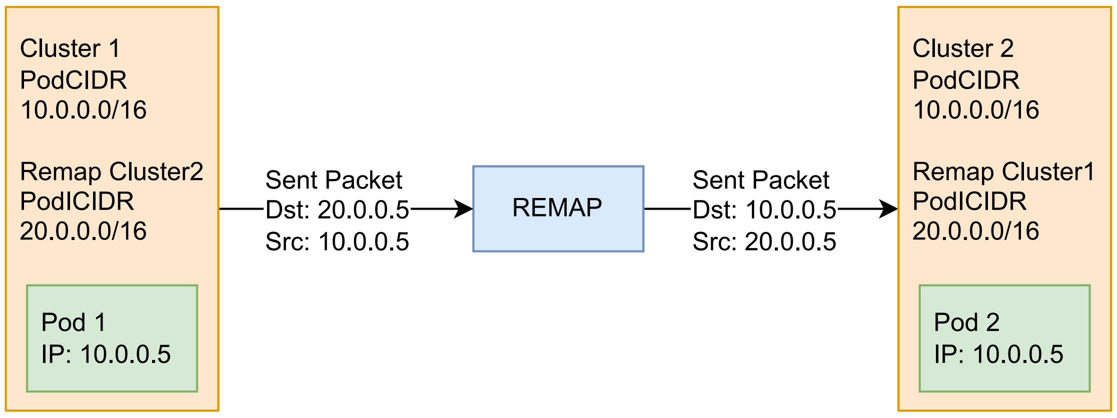


Figure 5.1: Remapping between 2 clusters with same PodCIDR

5.1.2 Exchanged information

A critical part to allow **remapping** are the exchanged information required to implement this mechanism. This will be a fundamental part to identify and evaluate the best design. The exchanged information, which could be useful to remap an IP is the locally used **PodCIDR** and how another cluster's **PodCIDR** has been remapped.

5.2 Architecture

This section presents in detail the two developed architectures. All the presented examples consider only 2 clusters for simplicity, but all the explained concepts can be applied to a more complex topology like the one in figure 5.2. Each design is based on natting (see section 4.2) and the focus will be on how to use this technology to get the best remapping mechanism.

5.2.1 Problem Area

The developed designs will be about the **cross-cluster area** of the Liqo network (see figure 5.2). The main component is the **gateway**, which acts as a funnel for all the network traffic directed to another peered cluster. Each gateway redirects the packets toward the correct cluster. The connections between gateways are called **peers**, each connection between two clusters has its own peer and each peer is independent from the others. They can be created and destroyed without affecting the connectivity towards other clusters. This is one of the fundamental concepts which stands behind the creation and deletion of **peerings**.

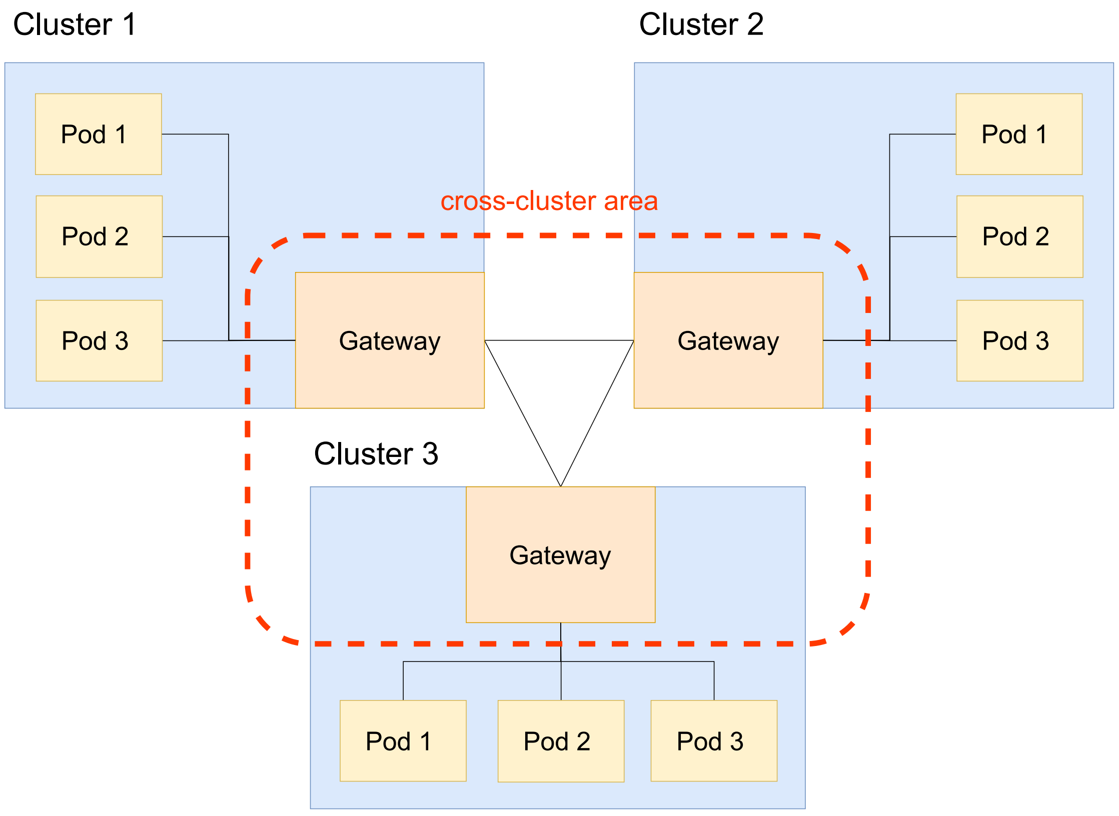


Figure 5.2: Liqo network cross-cluster area in a 3 clusters setup

5.2.2 Design SD

the name comes from the order in which the NAT rules are applied. To explain this design, will rely on the topology in figure 5.3. On the left, there is a cluster called **cluster 1** and on the right another one called **cluster 2**. **Cluster 1** uses as **PodCIDR** the range *40.0.0.0/16* and remap the **cluster 2**'s **PodCIDR** with *30.0.0.0/16*. **Cluster 2** presents the same **PodCIDR** (*40.0.0.0/16*) in order to have two overlapped IPs ranges, but it remaps the **cluster 1**'s **PodCIDR** with *20.0.0.0/16*. Of course, would be possible to use the same CIDR to remap each cluster, but two different CIDRs help to explain the scenario.

The presented images, show the path of a PING packet, going from a **cluster1** pod to a **cluster 2**'s pod having the same IP (*40.0.0.1*). The steps to make this possible are listed below:

1. **Cluster 1 - Redirect remapped IP towards the gateway**: This routing rule is used to allow packets with a remapped destination to reach the gateway. In the example, it redirects packets with *30.0.0.1* as destination IP to the gateway. If more than one pod with remapped IP has to be reached, all routing rules can be aggregated.

2. **Cluster 1 - SNAT**: This nat rule converts the source IP of each packet directed to **cluster 2**. The purpose of this conversion is to let **cluster 2** recognize these packets as incoming from **cluster 1**. Indeed, in the example, **cluster 2** has remapped **cluster 1** with *20.0.0.0/16* and the SNAT rule is converting all packets for **cluster 2** with a source IP which is part of *20.0.0.0/16*.

3. **Cluster 2 - DNAT**: this nat rule converts the destination IP of each packet incoming from **cluster 1**. When the pod inside **cluster 1** tries to contact the pod inside **cluster 2**, the first one needs to use as destination IP a remapped IP (in this case **30.0.0.1**). So when the packets reaches **cluster 2**, the remapped destination IP has no meaning anymore, and it has to be changed with a valid IP for **cluster 2**. So this nat rule converts *30.0.0.1* in *40.0.0.1*, which is the real destination pod IP.

4. **Cluster 2 - Redirect remapped IP towards the gateway**: This routing rule is similar to the first one. Indeed when the pod in **cluster 2**, has to answer to the sender, will generate a packet using as destination IP the one **cluster 2** used to remap **cluster 1**'s pod. So in the example **cluster 2**'s pod sends a packet using *20.0.0.1* as destination and the routing rule redirects this packet to the gateway.

Answer packets do not need any additional rules to reach **cluster 1**. Indeed the nat rules chain can be covered in reverse. However, this works only in the direction shown in figure 5.3, if the pod in cluster 2 wants to send a packet (which is not an answer) to the one in cluster 1, the same rules must be applied in the reverse order.

Exchanged information

In order to apply those rules, each cluster needs information about the other cluster. To create **SNAT** rules a cluster needs to know how it has been remapped by the peered clusters. Without this information is not possible to change the packets' source IPs to make them recognizable from the remote cluster.

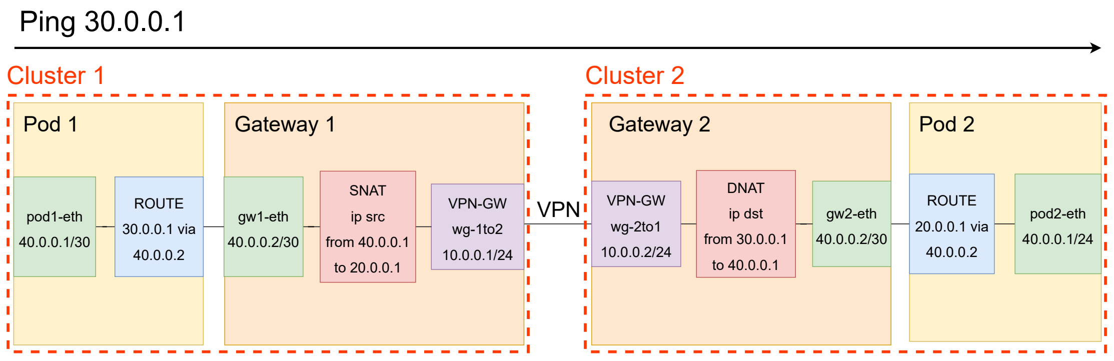

Ping 30.0.0.1
Cluster 1
Pod 1
pod1-eth 40.0.0.1/30
ROUTE 30.0.0.1 via 40.0.0.2
Gateway 1
gw1-eth 40.0.0.2/30
SNAT ip src from 40.0.0.1 to 20.0.0.1
VPN-GW wg-1to2 10.0.0.1/24
VPN
Cluster 2
Gateway 2
VPN-GW wg-2to1 10.0.0.2/24
DNAT ip dst from 30.0.0.1 to 40.0.0.1
gw2-eth 40.0.0.2/30
Pod 2
ROUTE 20.0.0.1 via 40.0.0.2
pod2-eth 40.0.0.1/24

Figure 5.3: Design SD architecture

5.2.3 Design DS

The name comes from the order in which the NAT rules are applied. To explain this design the topology in figure 5.4 will be used. The configuration is the same used for the **design SD** (see figure 5.3).

The presented image shows the path of a PING packet, going from a **cluster1** pod to a **cluster 2**'s pod having the same IP (*40.0.0.1*). The steps to make this possible are listed below:

1. **Cluster 1 - Redirect remapped IP towards the gateway**: This routing rule is used to allow packets with a remapped destination to reach the gateway. In the example, it redirects packets with *30.0.0.1* as destination IP to the gateway.

2. **Cluster 2 - DNAT**: this nat rule converts the destination IP of each packet going to **cluster 2**. When the pod inside **cluster 1** tries to contact the pod inside **cluster 2**, a remapped IP (in this case **30.0.0.1**) is used as destination IP. Differently from **design SD** (see figure 5.3) where the DNAT rule is applied in the destination cluster, here the operation is performed before, in **cluster 1**. In **design SD** is necessary to postpone DNAT rule because the destination remapped IP is used in the origin cluster to redirect a packet toward the correct remote cluster. Indeed. **Design SD** when the packet is redirected, it still has as destination IP the **remapped** IP. The information about how a packet has been remapped can be used to understand what is the destination cluster. In **Design DS** this information is not available because the **DNAT** converts the **remapped** destination IP before it can be used to select the destination cluster. So in **design SD** gateway can not redirect a packet to the correct cluster, this is a problem and will be discussed in the next section (see section 5.3) where it will be analyzed and solved. So when the packets reach **cluster 1**'s gateway, the remapped destination can be overwritten, and

it can be converted with an IP that is valid inside **cluster 2**. So this nat rule converts *30.0.0.1* in *40.0.0.1*, which is the real destination pod IP.

3. **Cluster 2 - SNAT**: This nat rule converts the source IP of each packet directed to **cluster 2**. The purpose of this conversion is to let **cluster 2** recognize these packets as incoming from **cluster 1**. Indeed, in the example, **cluster 2** has remapped **cluster 1** with *20.0.0.0/16* and the SNAT rule is converting all packets for **cluster 2** with a source IP which is part of *20.0.0.0/16*.

4. **Cluster 2 - Redirect remapped IP towards the gateway**: This routing rule is similar to the first one. Indeed when the pod in **cluster 2**, has to answer to the sender, will generate a packet using as destination IP the one **cluster 2** used to remap **cluster 1**'s pod. So in the example **cluster 2**'s pod sends packets using *20.0.0.1* as destination and the routing rule redirects these packets to the gateway.

Answer packets do not need any additional rules to reach **cluster 1**. Indeed the nat rules chain can be covered in reverse. However, this works only in the direction shown in figure 5.4, if the pod in cluster 2 wants to send a packet (which is not an answer) to the one in cluster 1, the same rules must be applied in the reverse order.

Exchanged information

Differently from **design SD** where each cluster needs to know how it has been remapped from the other one, in **design DS** this information is not necessary anymore. Anyway, each cluster needs to know which **PodCIDR** is used inside the peered clusters to apply the **DNAT** rule. Indeed when the remapped destination IP of a packet has to be converted, the origin cluster needs to know what are the valid IPs in the destination cluster.

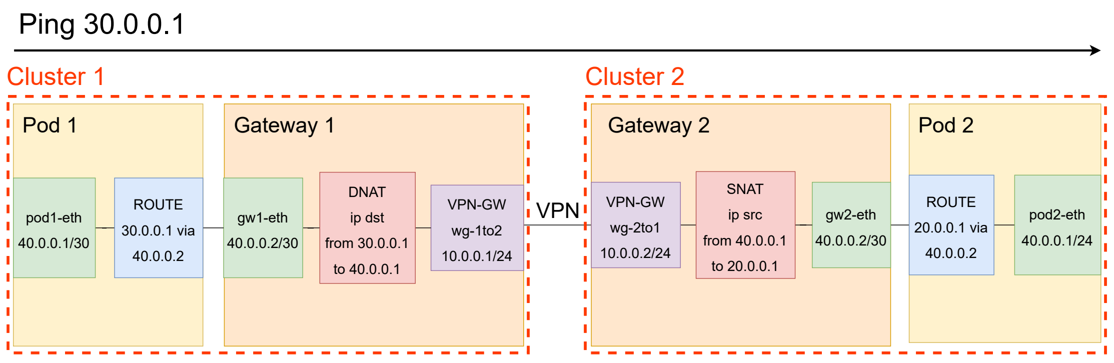

Ping 30.0.0.1
Cluster 1
Pod 1
pod1-eth 40.0.0.1/30
ROUTE 30.0.0.1 via 40.0.0.2
Gateway 1
gw1-eth 40.0.0.2/30
DNAT ip dst from 30.0.0.1 to 40.0.0.1
VPN-GW wg-1to2 10.0.0.1/24
VPN
Cluster 2
Gateway 2
VPN-GW wg-2to1 10.0.0.2/24
SNAT ip src from 40.0.0.1 to 20.0.0.1
gw2-eth 40.0.0.2/30
Pod 2
ROUTE 20.0.0.1 via 40.0.0.2
pod2-eth 40.0.0.1/24

Figure 5.4: Design DS architecture

5.2.4 Comparison

At first glance, it may seem that the two designs are the same in terms of exchanged information. **Design SD** needs to exchange remapped **PodCIDR**, while in **design DS** each cluster needs to know what are the real **PodCIDRs** in each cluster. Anyway, the second design is better and this is linked to detail about how Liqo works. Indeed Liqo needs to exchange the **NetworkConfig** resources (see section 3.5) which include information about **PodCIDRs** and **remapped PodCIDRs**. While **remapped PodCIDRs** are only used to setup nat rules, **PodCIDRs** are also used by other Liqo components. So the second design is the best because allows reducing the exchanged information, removing the **remapped PodCIDRs** from **NetworkConfig** resources.

5.3 Implementation

As explained in section 5.2.4, the **design DS** is the best one, thus it was the only one being considered for implementation. This section explains how the chosen design has been tested with a **proof of concept**, what choices have been made to implement it, and how problems have been solved.

5.3.1 Liqo NetNS

The implementation of the **gateway** is based on **linux network namespaces** (see section 4.1) and it is contained in the **liqo-gateway** component, which is a pod running in the Liqo namespace.

Usually, when a pod is instantiated, Kubernetes create a dedicated **network namespace** for that pod. However, Kubernetes allows using the **host** network namespace if specified in the pod resource. In Liqo this is useful, because allows applying the routing rules used to connect the **vxlan** (internal network part of Liqo), with the **cross-cluster** part of Liqo network, but it also creates a problem. The **cross-cluster** part of Liqo requires applying a set of **iptables rules** for every peer, which cannot be aggregated. In order to avoid the insertion of many **iptables rules** inside the host network namespace, the **liqo-gateway** component creates a dedicated **network namespace**. So the **liqo-gateway** component is a pod using the host **network namespace**, which creates autonomously another network-namespace, used to apply the nat rules (see figure 5.5).

5.3.2 Gateway redirection

In section 5.2.3 has been introduced a problem related to the design. After the packets are modified by the DNAT rule is not possible anymore to understand

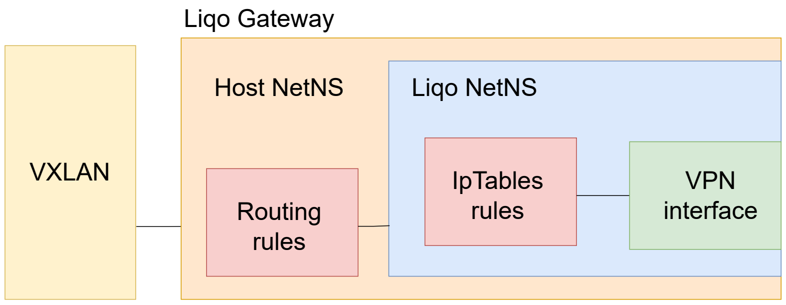


Figure 5.5: Liqo Gateway overview

what is the destination cluster. The information about how a packet has been remapped can be used to understand what is the destination cluster, but in **Design DS** this information is not available because the **DNAT** converts the **remapped** destination IP before it can be used to select the destination cluster. To solve this issue two alternatives have been found,

Multiple NetNS

The simplest solution is to have more than just one **Liqo NetNS** (see figure 5.6), in this way becomes easy to redirect the traffic. Indeed every **Liqo NetNS** has to redirect every packet coming from the **vxlan** to the VPN interface and vice versa. All the responsibilities about redirection towards the correct peered cluster become the responsibility of the **routing rules** inside the **host network namespace**.

Single NetNS

Another solution is to have a single **network namespace** with multiple **VPN interfaces** inside (see figure 5.7). In this case, the redirection towards a peered cluster is delegated to the **Liqo NetNS**, but the problem described in figure 5.4 persists. When a packet enters inside a **Liqo NetNS**, the first operation performed is a **DNAT** which overwrites the destination IP, so there is no possibility to redirect a packet. The solution to this problem is the **mark** rule. It is an iptables rule inserted inside the **mangle table**, which adds a label with a unique identifier to the packet. This identifier can be used to select the correct **VPN interface** used to reach the peered cluster. Given that **mark** rules can be placed in **prerouting** and **mangle table** has priority over **nat table**, it is possible to use the destination IP to mark a packet before the destination IP changes.

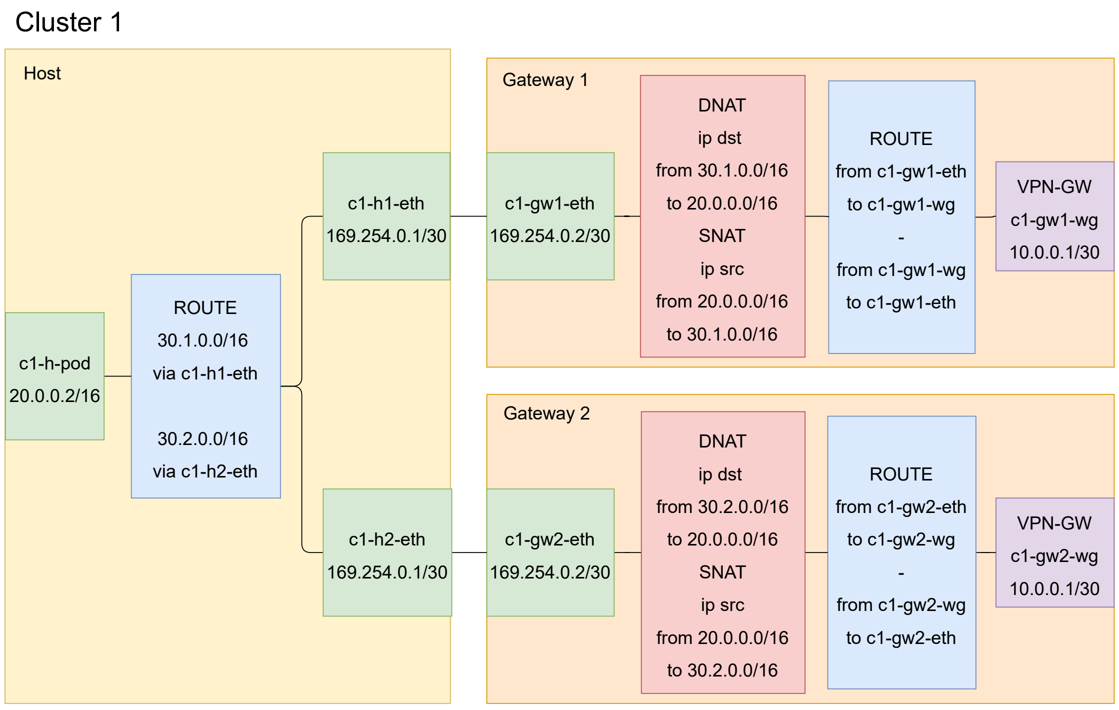

Cluster 1
Host
c1-h-pod 20.0.0.2/16
ROUTE 30.1.0.0/16 via c1-h1-eth 30.2.0.0/16 via c1-h2-eth
c1-h1-eth 169.254.0.1/30
c1-h2-eth 169.254.0.1/30
Gateway 1
c1-gw1-eth 169.254.0.2/30
DNAT ip dst from 30.1.0.0/16 to 20.0.0.0/16 SNAT ip src from 20.0.0.0/16 to 30.1.0.0/16
ROUTE from c1-gw1-eth to c1-gw1-wg - from c1-gw1-wg to c1-gw1-eth
VPN-GW c1-gw1-wg 10.0.0.1/30
Gateway 2
c1-gw2-eth 169.254.0.2/30
DNAT ip dst from 30.2.0.0/16 to 20.0.0.0/16 SNAT ip src from 20.0.0.0/16 to 30.2.0.0/16
ROUTE from c1-gw2-eth to c1-gw2-wg - from c1-gw2-wg to c1-gw2-eth
VPN-GW c1-gw2-wg 10.0.0.1/30

Figure 5.6: Implementation with multiple network namespaces

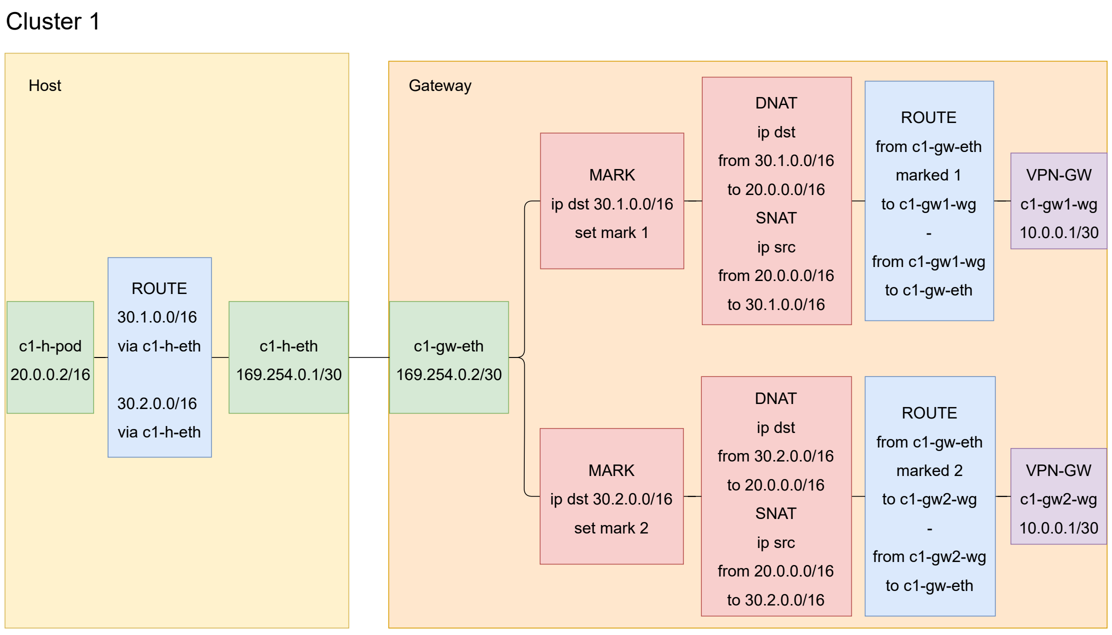

Cluster 1
Host
c1-h-pod 20.0.0.2/16
ROUTE 30.1.0.0/16 via c1-h-eth 30.2.0.0/16 via c1-h-eth
c1-h-eth 169.254.0.1/30
Gateway
c1-gw-eth 169.254.0.2/30
MARK ip dst 30.1.0.0/16 set mark 1
DNAT ip dst from 30.1.0.0/16 to 20.0.0.0/16 SNAT ip src from 20.0.0.0/16 to 30.1.0.0/16
ROUTE from c1-gw-eth marked 1 to c1-gw1-wg - from c1-gw1-wg to c1-gw-eth
VPN-GW c1-gw1-wg 10.0.0.1/30
MARK ip dst 30.2.0.0/16 set mark 2
DNAT ip dst from 30.2.0.0/16 to 20.0.0.0/16 SNAT ip src from 20.0.0.0/16 to 30.2.0.0/16
ROUTE from c1-gw-eth marked 2 to c1-gw2-wg - from c1-gw2-wg to c1-gw-eth
VPN-GW c1-gw2-wg 10.0.0.1/30

Figure 5.7: Implementation with a single network namespace

Comparison

Each solution takes advantages and disadvantages.

- **Single network namespace**
  - Advantages: Inside the host network namespace there is only one additional network namespace, it can be an advantage because it scales better and does noxt require to modify too much the host network namespace
  - Disadvantages: Every packet has to be "marked", this is an additional operation that is not needed in the other solution.
- **Multiple network namespace**
  - Advantages: Simplicity of the gateway, inside the Liqo NetNS, is performed only natting operations and the redirection is simpler than in the other solution.
  - *Disadvantages*: it does not scale well. Every peer needs a dedicated network namespace.

5.3.3 VPN interfaces

It is important to note another detail about these implementations.
In the first one (multiple namespaces), every namespace has its own VPN interface and this is mandatory. Also in the second implementation (single namespace) are present more interfaces, but it is just a way to simplify the design. It is possible another approach with a single VPN (see figure 5.8) where traffic is redirected. Of course, it is not a possible approach with all VPN technologies, but with **wireguard** (see section 4.3) is possible. So the second implementation makes the design more elastic and extensible.

Link local IP

Inside the gateway network namespace, a **link local** IP has been added to all interfaces. It is not mandatory but takes some advantages:

- It solves problems with ARP protocol, avoiding **proxy-arp** which caused the propagation of ARP messages through VPN tunnels.
- Gives the possibility to ping the other peered VPN interfaces. It helps with debugging and allows to perform a periodic check on clusters connection (see chapter 6)

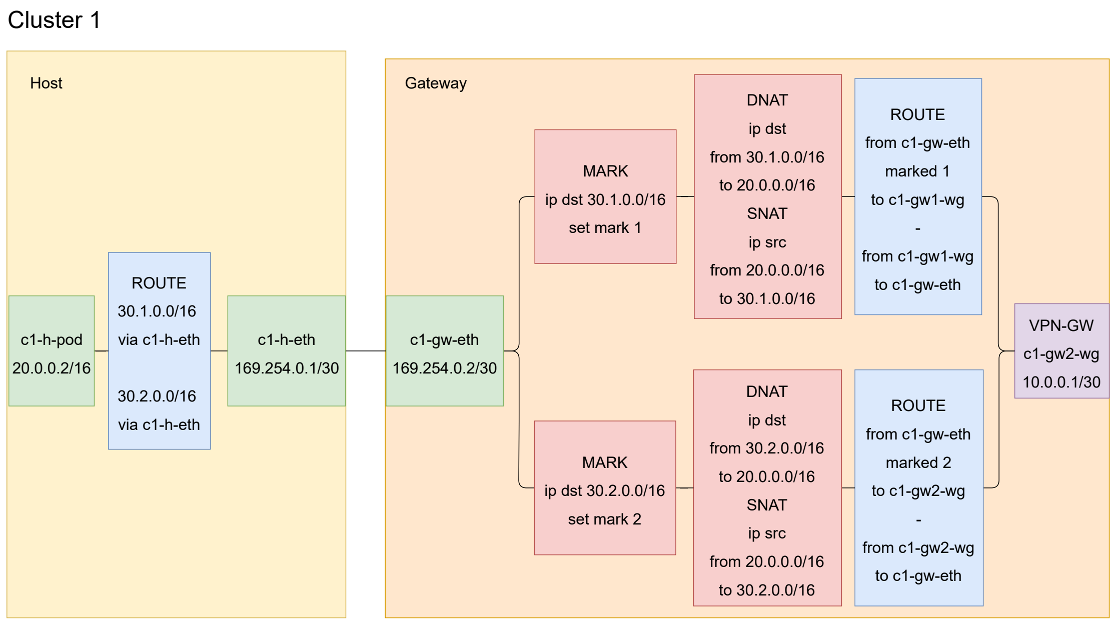


Figure 5.8: Implementation with a single Liqo Gateway and a single VPN interface

5.3.4 Conntrack

In Liqo, **NAT** rules are applied by the **Network Manager** (see chapter 3), which contains a dedicated controller. When two clusters peer all the nat rules have to be applied, but the controllers in each cluster are not synchronized so there is not a deterministic order. If a cluster tries to contact a peered one and the initialization phase has not ended yet, the received packets are automatically redirected to the **vxlan**. This redirection is not a problem, indeed when wrong packets reach the **vxlan** they are not accepted from any pod and they get lost. However when this redirection happens, **netfilter** setup a conntrack entry. After that, when all nat rules are applied, the conntrack makes it impossible for a packet to enter the nat chain. The only way to solve this issue is to delete the active conntracks, but it was not possible to implement this feature inside the code. So the best solution is to avoid the creation of this conntrack. A **DROP** iptables rule has been added in the **filter table**, which discards all the traffic not directed to the **vxlan**.

5.4 Conclusion

The described implementations work and both turned out to be a success. They allow reducing the exchange of information between clusters. As described in section 5.3.2 each implementation has its own advantages and disadvantages. They both are considered valid solutions but the best is considered the one based on a **single namespace** because **scalability** would be the best for the future of **Liqo**. A deeper evaluation of this chapter is contained in section 7.2, where have been discussed the integration and development of this implementation.

Chapter 6

Cross-Cluster Observability Design

This chapter explaines how cross-cluster network observability has been designed and implemented. Observability means being able to get information about the status of an entity or a process in real-time, to react in case of necessity. The reasons why the **cross-cluster network** observability can be considered useful are several and related to specific use cases, but they can be summarized in 2 categories:

- **Performance monitoring**: Getting information about clusters connections performance can be useful to evaluate how **Liqo** is behaving in a specific environment. For example, it helps to understand how much bandwidth offloaded applications are using in the communication between clusters, and to check that the obtained values respect the expected ones. If these values are not correct it may be a signal of some infrastructure or settings problems.
- **Fault detection**: Introducing a mechanism that gives information about the health of a connection, allowing monitoring systems (like **Prometheus** and **Grafana**) to send notifications in case of disruption.

6.1 The Problem

To monitor a system, metrics are fundamental. They are the data used to get information about the system's behavior. So the problems analyzed in this chapter will be related to these metrics. Two main problems can be found:

- **Get metrics**: what metrics can be used and how can be retrieved or created.
- **Expose metrics**: a way to expose metrics and to show the obtained results.

Metrics

Two types of metrics have been used. The first one is **Wireguard** metrics about each peer traffic:

- **Total received bytes**: the amount of data received by the VPN interface
- **Total sent bytes**: the amount of data transmitted by the VPN interface

These metrics are offered by wireguard Linux interface and have been exposed as they are. The second metrics type is the **Custom** metrics, which are generated by the **liqo-gateway** and will be discussed in this chapter. These metrics are:

- **The connection status**: if two peered clusters are able to cummunicate.
- **The latency in communication**: the round trip time between two peered clusters.

6.2 Overview

The purpose of the system that will be explained is to generate and update metrics related to **connectivity** and **latency**. The first one is a boolean value, it says if **connectivity** is working for a peer or not. The second one is a value indicating the **Round Trip Time** of packets transmission for a peer.

These metrics can be obtained using a **ping** mechanism. A cluster (having the **client** role) can send a **ping** packet and another cluster (with **server** role) can send a **pong** as answer. Using this mechanism is possible to check if there is connectivity between two clusters and calculate the latency in communications between them.

In **liqo-gateway** have been added two components (see figure 6.1):

- **Sender**: there is one for each peer. Its duty is to send **ping** packets to the other cluster. It doesn't receive answers from a peered cluster (this is a **receiver** task).
- **Receiver**: it is unique for each **liqo-gateway**. It can receive both **ping** or **pong** packets. If a **ping** is received it **sends** a **pong** to the peered cluster. If the received message is a **pong** it does not answer and calculates the metrics using the data transported by the packet (this data will be studied in depth in section 6.2.2).

Now that the main components have been introduced, is possible to explain the flow used by **liqo-gateways** to calculate the metrics. In this example, it is assumed to have two clusters peered with each other, called **Cluster1** and **Cluster2**. The steps performed are:

1. **Cluster1's sender** sends a **ping** message to the **Cluster2's receiver**.
2. **Cluster2's receiver** receives the **ping** message and recognizes it is a **ping**.
3. **Cluster2's receiver** sends a **pong** message to Cluster1's receiver.
4. **Cluster1's receiver** receives a **pong** and recognizes it as a **Cluster2's pong**.
5. **Cluster1's receiver** use the last received **pong** from **Cluster2** to generate metrics.

As explained in this process the **sender** never receives an answer and delegates it to the receiver. It does not act as a **client** which canonically sends and receives messages. This mechanism can sound strange and too complicated, but some technical motivations will be discussed in the sections about implementation (see section 5.3). Other details will be explained in the next sections: what data are transported by messages and how metrics are calculated in detail.

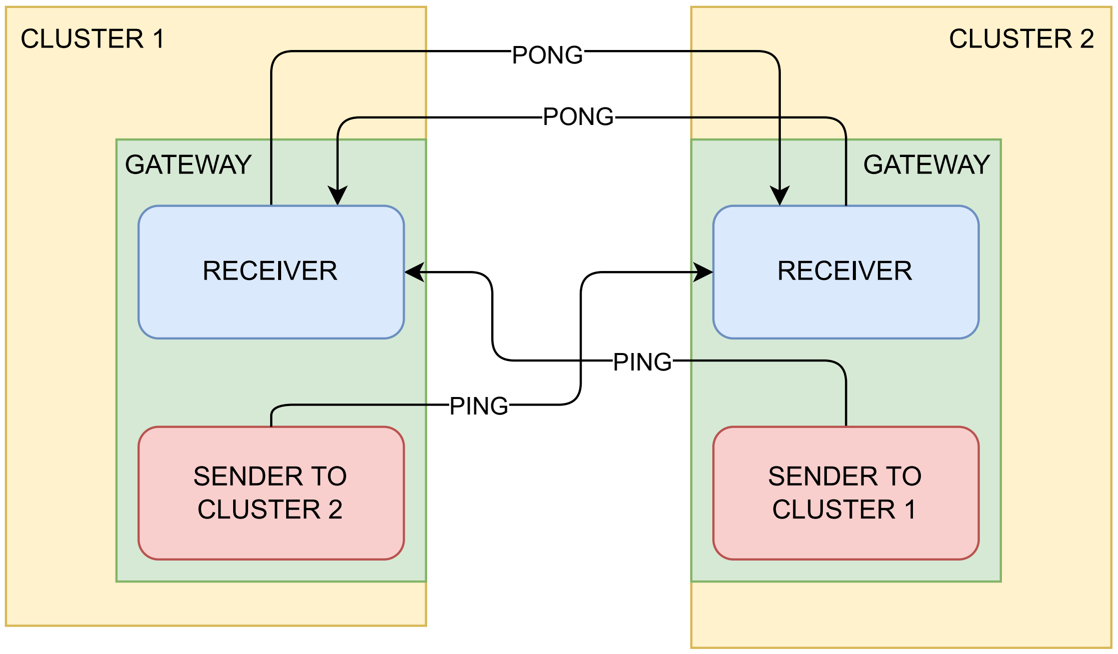


Figure 6.1: Connection checker overview

6.2.1 Implementation

The implementation is based on a **UDP ping**. To perform measures using **UDP packets** a protocol has been designed. **UDP** has been chosen instead of **TCP** to avoid **TCP** transmissions controls which would add a non-negligible error to the performed measures. Indeed, **TCP** guarantees that a packet is received and in case it is not delivered retransmits it. This mechanism is made by several steps which take time and increase the measured latency.

Protocol

The protocol is based on three information transported by packets. These are the data contained and their roles

- **Message type**: it can be **ping** or **pong**. It is used to understand if a packet is an answer to a previous request.
- **ClusterID**: it is a unique identifier for each cluster. It has a double function in case of ping or pong packets. If it is a **ping** it means that the destination of the packet is the cluster with the reported clusterID. Otherwise if it is a **pong** packet it means that it is an answer from that cluster.
- **Timestamp**: It is the timestamp generated when the **ping** packet has been generated. It is used to evaluate the latency.

6.2.2 ConnChecker

The implementation has been called **ConnChecker**. It is contained in a **GoLang** package called **conncheck**. It is a data structure containing the **receiver** and a map of **senders** (see code 6.1). It provides some methods to manage the object, listed down below.

Listing 6.1: ConnChecker data structure

```go
// ConnChecker is a struct that holds the receiver and senders.
type ConnChecker struct {
    receiver *Receiver
    // key is the target cluster ID.
    senders map[string]*Sender
    sm       sync.RWMutex
    conn     *net.UDPConn
}
```

NewConnChecker

It is a function used to create a new ConnChecker (see code 6.2.2). It returns a new object which can be used to start the **sender** and **receivers**.

```go
// NewConnChecker creates a new ConnChecker.
func NewConnChecker() (*ConnChecker, error) {
    addr := &net.UDPAddr{
        Port: port,
        IP:   net.ParseIP("0.0.0.0"),
    }
    conn, err := net.ListenUDP("udp", addr)
    if err != nil {
        return nil, fmt.Errorf("failed to listen on UDP socket %s : %w", addr, err)
    }
    klog.V(4).Infof("conncheck socket: listening on %s", addr)
    connChecker := ConnChecker{
        receiver: NewReceiver(conn),
        senders:  make(map[string]*Sender),
        conn:     conn,
    }
    return &connChecker, nil
}
```

RunReceiver

It is used to run the **Receiver**, The **Receiver** implementation will be discussed in the next sections. It is executed using a **go routine**, which allows running it in parallel.

RunReceiverDisconnectObserver

It runs the **Disconnect Observer** which will be discussed in section 6.3.4. It is executed using a **go routine**, which allows running it in parallel.

```go
// RunReceiverDisconnectObserver runs the receiver disconnect observer.
func (c *ConnChecker) RunReceiverDisconnectObserver() {
    c.receiver.RunDisconnectObserver()
}
```

AddAndRunSender

It creates a new sender for the specified remote cluster, adds it to the **map** inside the data structure, and runs it. It is executed using a **go routine**, which allows running it in parallel.

```go
// AddAndRunSender create a new sender and runs it.
func (c *ConnChecker) AddAndRunSender(clusterID, ip string,
    updateCallback UpdateFunc) {
    c.sm.Lock()
    if _, ok := c.senders[clusterID]; ok {
        c.sm.Unlock()
        klog.Infof("sender %s already exists", clusterID)
        return
    }

    ctxSender, cancelSender := context.WithCancel(context.Background
    ())
    c.senders[clusterID] = NewSender(ctxSender, clusterID,
    cancelSender, c.conn, ip)

    err := c.receiver.InitPeer(clusterID, updateCallback)
    if err != nil {
        c.sm.Unlock()
        klog.Errorf("failed to add redirect chan: %w", err)
    }

    klog.Infof("conncheck sender %s starting", clusterID)
    pingCallback := func(ctx context.Context) (done bool, err error)
    {
        err = c.senders[clusterID].SendPing(ctx)
        if err != nil {
            klog.Warningf("failed to send ping: %s", err)
        }
        return false, nil
    }
    c.sm.Unlock()

    // Ignore errors because only caused by context cancellation.
    _ = wait.PollImmediateInfiniteWithContext(ctxSender, PingInterval
    , pingCallback)

    klog.Infof("conncheck sender %s stopped", clusterID)
}
```

DelAndStopSender

It stops a sender related to a remote cluster and removes it from the **map** contained in the data structure.

```go
// DelAndStopSender stops and deletes a sender. If sender has been
    already stoped and deleted is a no-op function.
func (c *ConnChecker) DelAndStopSender(clusterID string) {
    c.sm.Lock()
    defer c.sm.Unlock()

    c.receiver.m.Lock()
    defer c.receiver.m.Unlock()

    if _, ok := c.senders[clusterID]; ok {
        c.senders[clusterID].cancel()
        delete(c.senders, clusterID)
    }
    delete(c.receiver.peers, clusterID)
}
```

GetLatency

It allows retrieving the last measured latency related to a remote cluster.

```go
// GetLatency returns the latency with clusterID.
func (c *ConnChecker) GetLatency(clusterID string) (time.Duration,
    error) {
    c.receiver.m.RLock()
    defer c.receiver.m.RUnlock()
    if peer, ok := c.receiver.peers[clusterID]; ok {
        return peer.latency, nil
    }
    return 0, fmt.Errorf("sender %s not found", clusterID)
}
```

GetConnected

It allows getting a boolean indicating if the connection between the local cluster and the specified remote cluster is working.

```go
// GetConnected returns the connection status with clusterID.
func (c *ConnChecker) GetConnected(clusterID string) (bool, error) {
    c.receiver.m.RLock()
    defer c.receiver.m.RUnlock()
    if peer, ok := c.receiver.peers[clusterID]; ok {
        return peer.connected, nil
    }
    return false, fmt.Errorf("sender %s not found", clusterID)
}
```

6.3 Messages

Exchanged messages are defined like a **GoLang** data structure called **Msg** (see code 6.3). It contains:

- **ClusterID**: which is the remote cluster **clusterID**, an identifier for each cluster.
- **MsgType**: it is used to distinguish **ping** messages (sent by senders) and **pong** messages (sent by receivers).
- **TimeStamp**: it contains the timestamp generated from the sender when the **ping** packet is sent.

Msg data structure contains some annotations which allow to **marshal** and **unmarshal** the **Msg** objects.

```go
// Msg represents a message sent between two nodes.
type Msg struct {
	ClusterID string    `json:"clusterID"`
	MsgType   MsgTypes  `json:"msgType"`
	TimeStamp time.Time `json:"timeStamp"`
}

// MsgTypes represents the type of a message.
type MsgTypes string

const (
	// PING is the type of a ping message.
	PING MsgTypes = "PING"
	// PONG is the type of a pong message.
	PONG MsgTypes = "PONG"
)
```

6.3.1 Sender

ConnCheckers contains a **Sender** for each peered remote cluster. Each sender contains:

- **clusterID**: the clusterID of the remote cluster, target of the **ping** messages.
- **raddr**: the IP address of the remote cluster, the target of the **ping** messages.

- **cancel**: a callback used to stop the sender before it is removed.
- **conn**: a variable representing the socket used to send messages.

The sender can be created and run using the **AddAndRunSender** method (see section 6.2.2).

Listing 6.2: Sender data structure

```go
// Sender is a sender for the conncheck server.
type Sender struct {
    clusterID string
    cancel    func()
    conn      *net.UDPConn
    raddr     net.UDPAddr
}
```

NewSender

It is the function used to create and initialize a new sender for a remote cluster (see code 6.3). It is important to notice that this function does not initialize a new **socket** when it is called, but receives a pointer to an already existing socket (parameter **conn**). This means that each sender uses the same socket, which is the same one used by the local receiver. The advantages of this approach are explained in section 6.3.2

Listing 6.3: NewSender function

```go
// NewSender creates a new conncheck sender.
func NewSender(ctx context.Context, clusterID string, cancel func(),
   conn *net.UDPConn, ip string) *Sender {
    return &Sender{
        clusterID: clusterID,
        cancel:    cancel,
        conn:      conn,
        raddr:     net.UDPAddr{IP: net.ParseIP(ip), Port: port},
    }
}
```

SendPing

It is the method used by Sender objects to send **ping** messages (see code 6.4) in loop for each **PingInterval**. It generates the **message** and sends it using the **UDP socket**.

Listing 6.4: Sender's PingInterval method

```go
// SendPing sends a PING message to the given address.
func (s *Sender) SendPing(ctx context.Context) error {
    msgOut := Msg{ClusterID: s.clusterID, MsgType: PING, TimeStamp:
   time.Now()}
    b, err := json.Marshal(msgOut)
    if err != nil {
        return fmt.Errorf("conncheck sender: failed to marshal msg: %
   w", err)
    }
    _, err = s.conn.WriteToUDP(b, &s.raddr)
    if err != nil {
        return fmt.Errorf("conncheck sender: failed to write to %s: %
   w", s.raddr.String(), err)
    }
    klog.V(8).Infof("conncheck sender: sent a PING -> %s", msgOut)
    return nil
}
```

6.3.2 Receiver

Every **ConnChecker** contains only one **receiver**. The Receiver is a **GoLang** data structure (see code 6.5). It contains a **map** called **peers**, the keys are the remote clusters' **clusterIDs** and the values are **Peer** objects. **Peer** is a data structure containing the last evaluated metrics and other information about a remote cluster. It contains:

- **connected**: metric about the connectivity status of a peer with a remote cluster. It is a **boolean** and its value is **true** if the communication between clusters is possible.
- **latency**: metric about latency time between the local cluster and a peered one.
- **lastReceivedTimestamp**: timestamp of the last received **ping** message. It is used by the **Disconnect Observer** (see section 6.3.4).
- **updateCallback**: it is a function used to update a peer status inside the **TunnelEndpoint CRD** (see section 6.3.5).

The receiver can be run using the **RunReceiver** method (see section 6.2.2).

Listing 6.5: Receiver and Peer data structures

```go
// Peer represents a peer.
type Peer struct {
    connected bool
    latency   time.Duration
    // lastReceivedTimestamp is the timestamp when the last received
   PING has been sent.
    lastReceivedTimestamp time.Time
    updateCallback        UpdateFunc
}

// Receiver is a receiver for conncheck messages.
type Receiver struct {
    peers map[string]*Peer
    m     sync.RWMutex
    buff  []byte
    conn  *net.UDPConn
}
```

Run

The **Run** method is used to start the **Receiver** (see code 6.6). This is the core of the **ConnChecker** mechanism. An important detail about the Receiver is that it uses only one socket to listen for incoming packets. As explained in previous sections, senders do not listen for answers packets, indeed all incoming packets (**ping** and **pong**) are received and managed by the receiver. The reason behind this choice is to **use only one socket**. Every cluster runs only one receiver per **liqo-gateway**, which means that only one socket needs to be opened. Furthermore, every sender uses the same socket, which is the same used by receivers (see section 6.3.2). The advantage of this approach is **Scalability**. In **Liqo** a cluster is able to peer with a huge number of clusters. Having to open a new socket for each peered cluster could become a problem. So being able to use just one socket can help to scale.

Listing 6.6: Receiver's Run method

```go
// Run starts the receiver.
func (r *Receiver) Run() {
    klog.V(8).Infof("conncheck receiver: starting")
    for {
        n, raddr, err := r.conn.ReadFromUDP(r.buff)
        if err != nil {
            klog.Errorf("conncheck receiver: failed to read from %s:
   %w", raddr.String(), err)
            continue
        }
        msgr := &Msg{}
        err = json.Unmarshal(r.buff[:n], msgr)
        if err != nil {
            klog.Errorf("conncheck receiver: failed to unmarshal msg:
    %w", err)
            continue
        }
        klog.V(9).Infof("conncheck receiver: received a msg -> %s", msgr)
        switch msgr.MsgType {
        case PING:
            klog.V(8).Infof("conncheck receiver: received a PING %s -> %s", raddr, msgr)
            err = r.SendPong(raddr, msgr)
        case PONG:
            klog.V(8).Infof("conncheck receiver: received a PONG from %s -> %s", raddr, msgr)
            err = r.ReceivePong(msgr)
        }
        if err != nil {
            klog.Errorf("conncheck receiver: %v", err)
        }
    }
}
```

6.3.3 Generate metrics

The metrics generation is performed by **Receiver**. When a **pong** packet is received (see code 6.6), the **ReceivePong** method is called (see code 6.7). The received message contains:

- the **clusterID** of the remote cluster which is sending the message itself (it allows to distinguish **pong** messages from different clusters)
- the **TimeStamp**, which is the moment in which the local cluster sent the **ping** message to which the received message is answering.

In order to discard **out of order** packets (because **UDP** does not guarantee the correct order for received packets) **TimeStamp** and **lastReceivedTimestamp** are compared. If the just received **TimeStamp** is lower than the last received it means that the packet is an old packet and it has to be discarded.

If the packet is considered valid, metrics can be evaluated. The **latency** can be obtained from the difference between the received timestamp (**TimeStamp variable**) and the current timestamp. It is possible to set to true the **connected** variable. At the end, metrics are propagated to the **TunnelEndpoint** resource (see section 6.3.5).

Listing 6.7: Receiver's ReceivePong method

```
// ReceivePong receives a PONG message.
func (r *Receiver) ReceivePong(msg *Msg) error {
    r.m.Lock()
    defer r.m.Unlock()
    if peer, ok := r.peers[msg.ClusterID]; ok {
        if msg.TimeStamp.Before(peer.lastReceivedTimestamp) {
            klog.V(8).Infof("dropped a PONG message from %s because
out-of-order", msg.ClusterID)
            return nil
        }
        now := time.Now()
        peer.lastReceivedTimestamp = msg.TimeStamp
        peer.latency = now.Sub(msg.TimeStamp)
        peer.connected = true

        err := peer.updateCallback(true, peer.latency, now)
        if err != nil {
            return fmt.Errorf("failed to update peer %s: %w", msg.
ClusterID, err)
        }
        return nil
    }
    return fmt.Errorf("%s sender has not been initialized", msg.
ClusterID)
}
```

6.3.4 Disconnect Observer

The **Disconnect Observer** is part of the **receiver**. It is a method of the Receiver data structure called **RunDisconnectObserver** (see code 6.8). It is launched in parallel with the receiver's Run method. If the Run method purpose is to update the metrics in case of success, the disconnect observer has to update them in case of failure. The mechanism used to notice a failure is based on the **lastReceivedTimestamp**. If a defined threshold in terms of time has passed from the last received ping it means that connectivity is not working anymore and that a failure has to be notified. So two variables have been defined:

- **PingLossThreashold**: The number of lost packets after a failure has to be notified.
- **PingInterval**: The same variable used by senders to periodically send a ping.

The product of these two variables gives a time interval that can be used to check if too much time has passed from the last received valid **pong** message. This check is repeated periodically. In case of failure, the **connected** variable is set to false and the **latency** becomes 0. In the end, metrics are propagated to the **TunnelEndpoint** resource (see section 6.3.5).

Listing 6.8: Receiver's RunDisconnectObserver method

```go
// RunDisconnectObserver starts the disconnect observer.
func (r *Receiver) RunDisconnectObserver() {
    klog.V(9).Infof("conncheck receiver disconnect checker: starting")
    // Ignore errors because only caused by context cancellation.
    _ = wait.PollImmediateInfiniteWithContext(context.Background(), time.Duration(PingLossThreshold)*PingInterval/10,
        func(ctx context.Context) (done bool, err error) {
            r.m.Lock()
            defer r.m.Unlock()
            for id, peer := range r.peers {
                if time.Since(peer.lastReceivedTimestamp.Add(peer.latency)) <= PingInterval*time.Duration(PingLossThreshold) {
                    continue
                }
                klog.V(8).Infof("conncheck receiver: %s unreachable", id)
                peer.connected = false
                peer.latency = 0
                err := peer.updateCallback(false, 0, time.Time{})
                if err != nil {
                    klog.Errorf("conncheck receiver: failed to update peer %s: %s", peer.lastReceivedTimestamp, err)
                }
            }
            return false, nil
        })
}
```

6.3.5 TunnelEndpoint update

When metrics are evaluated, they need to be saved somewhere in order to be available to be shown. For this reason, some fields have been added to the **TunnelEndpoint** custom resource's status. In particular, TunnelEndpoint's status contains a **connection** section. The following fields have been added:

- **status**: the connectivity status, possible values are **Connected**, **Connecting** or **Error**.
- **statusMessage**: A message explaining the connection status.
- **latency**: contains information about current latency
  - **value**: the last evaluated latency.
  - **timestamp**: the timestamp of the moment in which the last latency has been evaluated.

The propagation of metrics inside the TunnelEndpoint is performed by the **updateCallback** function (see section 6.3.2). This function is generated using **forgeConncheckUpdateStatus** function (see code 6.9). The CRD output showed with **liqoctl get tunnelendpoint -o wide** has been modified to simplify the visualization of the metrics.

Listing 6.9: TunnelController's forgeConncheckUpdateStatus method

```go
func (tc *TunnelController) forgeConncheckUpdateStatus(ctx context.Context, req ctrl.Request) conncheck.UpdateFunc {
    return func(connected bool, latency time.Duration, timestamp time.Time) error {
        var tep = new(netv1alpha1.TunnelEndpoint)
        if err := tc.Get(ctx, req.NamespacedName, tep); err != nil && !k8sApiErrors.IsNotFound(err) {
            return fmt.Errorf("unable to fetch resource %s: %w", req.String(), err)
        }
        conn := tep.Status.Connection
        if connected {
            conn.Status = netv1alpha1.Connected
            conn.StatusMessage = netv1alpha1.ConnectedMessage
        } else {
            conn.Status = netv1alpha1.ConnectionError
            conn.StatusMessage = netv1alpha1.ConnectionErrorMessage
        }
        if tep.Status.Connection.Status != conn.Status || tep.Status.Connection.StatusMessage != conn.StatusMessage ||
            timestamp.Sub(tep.Status.Connection.Latency.Timestamp.Time) > tc.updateStatusInterval {
            if tep.Status.Connection.Status != conn.Status || tep.Status.Connection.StatusMessage != conn.StatusMessage {
                klog.Infof("%s -> changing status to %s %q",
                    tep.Spec.ClusterIdentity, conn.Status, conn.StatusMessage)
            }
            conn.Latency = netv1alpha1.ConnectionLatency{
                Value:     liqonetutils.FormatLatency(latency),
                Timestamp: metav1.Time{Time: timestamp},
            }
            tep.Status.Connection = conn
            if err := tc.Client.Status().Update(ctx, tep); err != nil {
                return fmt.Error("unable to update resource %s: %w", req.String(), err)
            }
        }
        return nil
    }
}
```

6.3.6 Prometheus Metrics

Prometheus

Prometheus is a free software application used for event monitoring and alerting. It records real-time metrics in a database. Prometheus data is stored in the form of metrics, with each metric having a name that is used for referencing and querying it. Each metric can be enriched by an arbitrary number of key/value pairs called labels, containing information about the metric's source.

Metrics exposition

As explained in the previous section, evaluated metrics have been propagated in the **TunnelEndpoint** resources, but this is not enough. Indeed metrics need to be **exposed** to other microservices. This allows external entities to be aware of the **cross-cluster connection** state for each peer. The retrieved data can be analyzed and reworked to obtain further information.

Metrics have been exposed using GoLang **prometheus exporter**, to expose them in **prometheus** format. A specific implementation of the **prometheus exporter** can be developed for each **VPN driver**. This allows having different metrics, for different VPN drivers. At the moment **Liqo** just supports **Wireguard** as VPN driver, so at the moment the only implementation available is related to it. As explained in the previous sections two types of metrics can be exposed. The first includes metrics provided by the **VPN interface**. For example, **Wireguard** can save some statistics inside the interface's configuration file. These data have been used to expose these metrics

- **liqo_peer_receive_bytes_total**: the total amount of received bytes.
- **liqo_peer_transmit_bytes_total**: the total amount of transmitted bytes

The second type of metric is the one not related to the VPN interface and they are common to every VPN driver. They are the metrics evaluated by the **ConnChecker** and have been exposed using these names:

- **liqo_peer_is_connected**: it is an integer value, which can be 1 or 0. If the value is 1 the connection is working successfully, otherwise, it means that a failure has been detected.
- **liqo_peer_latency_us**: it is the latency (in microseconds).

These metrics are related to each peer, so when they are exposed they need to be distinguished according to the peer to which they refer. To do this, a set of labels have been attached to all metrics. These labels are:

- **clusterID**: The ID of the remote cluster
- **clusterName**: The name of the remote cluster
- **device**: The name of the VPN interface used to establish the connection
- **driver**: The name of the VPN river used (eg. Wireguard)

How metrics are scraped will depend on how is operating the Prometheus server. Liqo presumes that the Prometheus Operator is being used to run Prometheus, providing a ServiceMonitor resource for each component. Metrics are disabled by default. In order to allow Prometheus to scrape metrics from the Liqo components, is necessary to set the **–enable-metrics** liqoctl flag during installation. This flag enables the metrics exposition and the ServiceMonitor resources creation. A ServiceMonitor is a CRD offered by the Prometheus Operator, it is used to allow the Prometheus server to find the Service exposing the metrics.

Chapter 7

Evaluation

This chapter discusses and evaluates the results obtained in each part of the thesis. Will be explained how obtained results can be useful and be exploited to increase the user experience of **Liqo**

7.1 Network Design

In chapter 5 two possible network designs have been presented. As explained in section 5.2.4 the **design SD** is considered the best. However a detail has not been discussed yet about it, the amount of work necessary to switch to this design. At the moment **Liqo** uses the **first design** which has been implemented and took years to be stable. So the implementation of a new design is not simple work and requires a not negligible amount of time and human resources. This is the trade-off that has to be considered before implementing the new design. On one hand, there is less information that has to be exchanged and a more stable peering phase. On the other hand, the **liqo network-manager** needs to be rewritten and designed, this means a lot of time (maybe more than a year) to have a working and stable new **network-manager** supporting the **design DS**. Furthermore, the improvement taken by the new design would not justify a similar workload. So have been decided to not implement the new design and keep the legacy one.

Not always the solution with better performance is the best one, the time of a team is precious and must be exploited to create new features and increase the value of a product. A change like the one described before would be invisible and non-perceived by the users using **Liqo**.

7.2 Observability Design

Chapter 6 has described how in **Liqo** has been designed and implemented a mechanism to check and monitor **the cross-cluster network** area. In order to do this the **ConnChecker** has been implemented and the **VPN driver**, used by Liqo, has been modified to support metrics exposition (customizable for each VPN driver).

7.2.1 Grafana

Section 6.3.6 discussed the metrics exposed. To test the exposed metrics a sample Grafana dashboard has been built. It allows the monitoring of the network interconnection for an arbitrary number of Liqo peerings. Grafana is a multi-platform open-source analytics and interactive visualization web application. It provides charts, graphs, and alerts for the web when connected to supported data sources. The provided dashboard includes an **overview** section presenting the overall cross-cluster throughput, followed by detailed **per-peering** information (see figure 7.1). In particular the **overview** section includes:

- **Connected peers amount**: They are the peers which are receiving correctly **pong** packets in answer to **ping** packets sent to the remote cluster
- **Disconnected peers amount**: They are the peers which are not receiving **pong** packets in answer to **ping** packets sent to the remote cluster
- **Received throughput**: the received data throughput of all peered clusters shown stacked
- **Transmitted throughput**: the transmitted data throughput of all peered clusters shown stacked

The **per-peering** sections are dedicated to a single peer and are generated automatically when new peers are established:

- **Connectivity**: connection status
- **Throughput**: received and transmitted data throughput
- **Latency**
- **Average Latency**

Througput in every graph has been obtained evaluating the **liqo_peer_receive_bytes_total** and **liqo_peer_transmit_bytes_total** metrics rate. The interval rate is the one defined by the **Prometheus Server**.

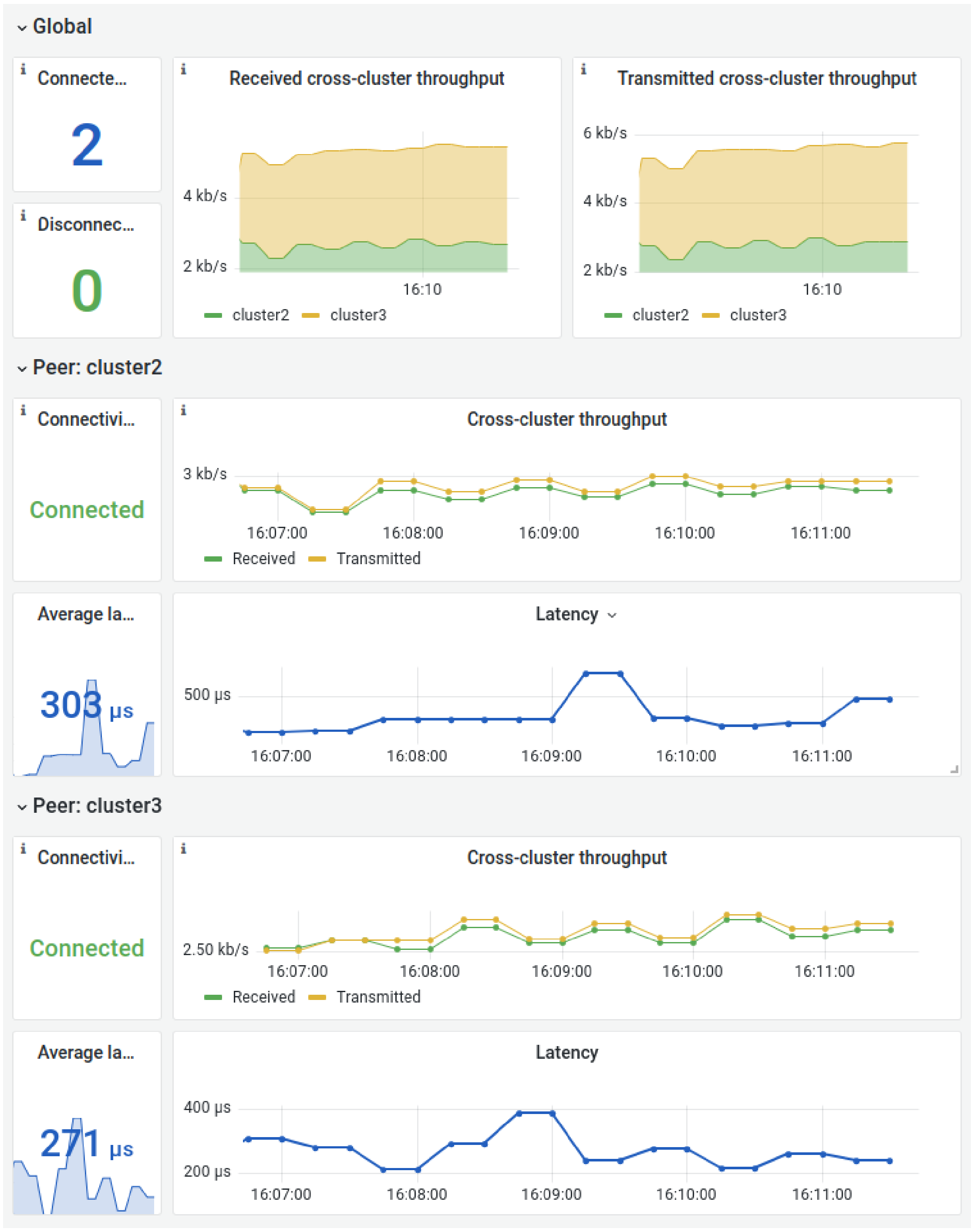


Figure 7.1: Liqo cross-cluster network grafana dashboard

7.2.2 Liqoctl Status

To give to the users a way to check information about connectivity without having to enable metrics or watch a TunnelEndpoint resource, the **status** command of **liqoctl** has been extended. Furthermore, other features have been added to this command, which are not strictly related to peer monitoring. The majority of **liqoctl status** command has been rewritten from scratch, and now it allows to have a better overview about a Liqo installation.

Local

This section is about the **liqoctl status** command (see list 7.1). It checks the existence of the Liqo namespace and checks the readiness of all Liqo components. A further section has been added in the command output, about **local information**. These are details about the Liqo installation. It means that they are information not related to remote clusters peered. An output of this command can be seen in list 7.1.

Listing 7.1: liqoctl status output

```
> ./liqoctl status
Namespace existence check
INFO  liqo control plane namespace liqo exists

Liqo control plane check
INFO  control plane pods are up and running

Local Cluster Information
Cluster Identity
      Cluster ID:   3c3c94e3-07c0-4787-ac2c-de12788d41cb
      Cluster Name: cluster1
      Cluster Labels
          liqo.io/provider: kind
  Network
      Pod CIDR:       10.112.0.0/16
      Service CIDR:   10.111.0.0/16
      External CIDR:  10.110.0.0/16
```

Peer

This section is about the new **liqoctl status peer** command (see list 7.1). It does not provide only network information about a peer. It is possible to use as arguments the name of the peered clusters to select the ones whose information has to be printed. Otherwise, if no argument is provided the information about all peered clusters is printed. It is also possible to increase the verbosity of the command using the –**verbose** flag. A verbose output of this command can be seen in list 7.2

Listing 7.2: liqoctl status peer output

```
> liqoctl status peer cluster2 cluster3 --verbose
Peered Cluster Information
  cluster2 - a1c73050-d1ea-4dee-84d2-e548e5242298
      Type: InBand
      Direction
          Outgoing: Established
          Incoming: Established
      Authentication
          Status:          Established
          Auth URL:        https://10.108.0.3:443
          Auth Proxy URL: http://10.108.0.2:8118
      Network
          Status: Established
          Remote CIDRs
              Original
                  Pod CIDR:       10.112.0.0/16
                  External CIDR: 10.110.0.0/16
              Remapped - how "cluster1" remapped "cluster2"
                  Pod CIDR:       10.113.0.0/16
                  External CIDR: 10.108.0.0/16
          Local CIDRs
              Original
                  Pod CIDR:       10.112.0.0/16
                  External CIDR: 10.110.0.0/16
              Remapped - how "cluster1" has been remapped by "
    cluster2"
                  Pod CIDR:       10.113.0.0/16
                  External CIDR: 10.108.0.0/16
          VPN Connection
              Status:  Connected - VPN connection established
              Latency: 301us
              Gateway IPs
                  Local:  172.18.0.3:30987
                  Remote: 172.18.0.4:32084
      Resources
          Total acquired - resources offered by "cluster2" to "
    cluster1"
              cpu:                  2190m
              memory:               3.13GiB
              pods:                 22
              ephemeral-storage: 40.95GiB
          Total shared - resources offered by "cluster1" to "cluster2
    "
              cpu:                  1015m
```

```
                  memory:             1.36GiB
                  pods:               11
                  ephemeral-storage:  20.48GiB
  cluster3 - c07ad6c6-443a-44a4-bf2a-fdbf696309ce
      Type: InBand
      Direction
          Outgoing: Established
          Incoming: Established
      Authentication
          Status:         Established
          Auth URL:       https://10.114.0.3:443
          Auth Proxy URL: http://10.114.0.2:8118
      Network
          Status: Established
          Remote CIDRs
              Original
                  Pod CIDR:      10.112.0.0/16
                  External CIDR: 10.110.0.0/16
              Remapped - how "cluster1" remapped "cluster3"
                  Pod CIDR:      10.109.0.0/16
                  External CIDR: 10.114.0.0/16
          Local CIDRs
              Original
                  Pod CIDR:      10.112.0.0/16
                  External CIDR: 10.110.0.0/16
              Remapped - how "cluster1" has been remapped by "cluster3"
                  Pod CIDR:      10.113.0.0/16
                  External CIDR: 10.108.0.0/16
          VPN Connection
              Status:  Connected - VPN connection established
              Latency: 211us
              Gateway IPs
                  Local:  172.18.0.3:30987
                  Remote: 172.18.0.6:31350
      Resources
          Total acquired - resources offered by "cluster3" to "cluster1"
              cpu:                3285m
              memory:             4.69GiB
              pods:               33
              ephemeral-storage:  61.43GiB
          Total shared - resources offered by "cluster1" to "cluster3"
              cpu:                1015m
              memory:             1.36GiB
              pods:               11
              ephemeral-storage:  20.48GiB
```

7.2.3 Performance

This section compares the performance of the old and new versions of Liqo. In particular, the resource consumption of the versions **0.5.4** and **0.6.0** (the first containing the thesis work) of Liqo have been observed. The comparison aims to evaluate the impact of the **connchecker** inside the liqo-gateway. To obtain statistics about **CPU**, **memory**, and **traffic**, a cluster has been peered with two other clusters. During the period in which statistics have been acquired the clusters stayed in IDLE. It means that the tests were conducted without forcing traffic through the liqo-gateway, to observe just the effects of the connchecker.

Table 7.1 and 7.2 graphs contain the statistics about each version. All statistics are higher in version **0.6.0** and this is an expected result. But the values obtained from the old and the new version of Liqo differ slightly. The average CPU and memory used differ by a negligible amount. Same for the throughput, which slightly increases in the new version due to the periodic ping performed by the connchecker.

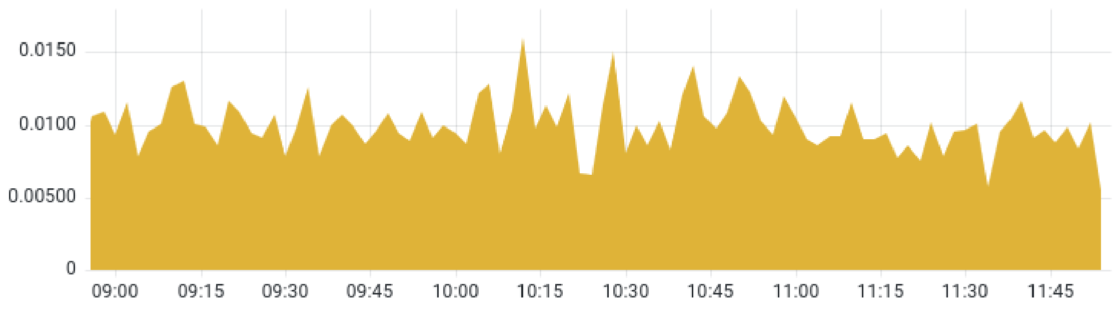
So, the performance evaluation is a success, new features have been added to Liqo and this has not an impact on consumed resources.

	Min	**Max**	**Average**
CPU percentage used	0.005%	0.02%	0.01%
Memory used	18.9MiB	20.5Mib	18.9MiB
Received data	0.09kB/s	2.3kB/s	0.52kB/s
Transmitted data	0.09kB/s	5.68kB/s	0.66kB/s

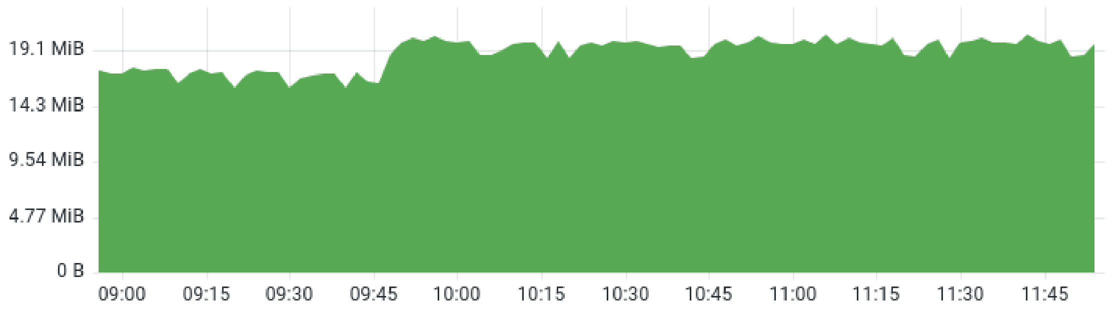
Table 7.1: Liqo-gateway performance statistics in Liqo 0.5.4

	Min	**Max**	**Average**
CPU percentage used	0.02%	0.05%	0.03%
Memory used	20.2MiB	25.7Mib	23.4MiB
Received data	0.56kB/s	3.87kB/s	1.56kB/s
Transmitted data	0.53kB/s	6.95kB/s	1.65kB/s

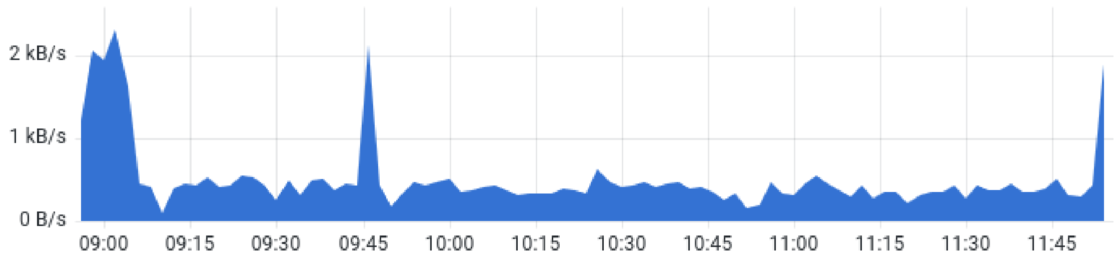
Table 7.2: Liqo-gateway performance statistics in Liqo 0.6.0



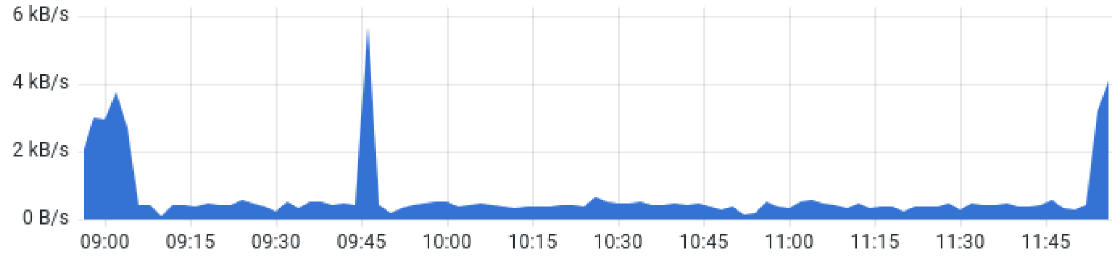
CPU consumption



Memory consumption

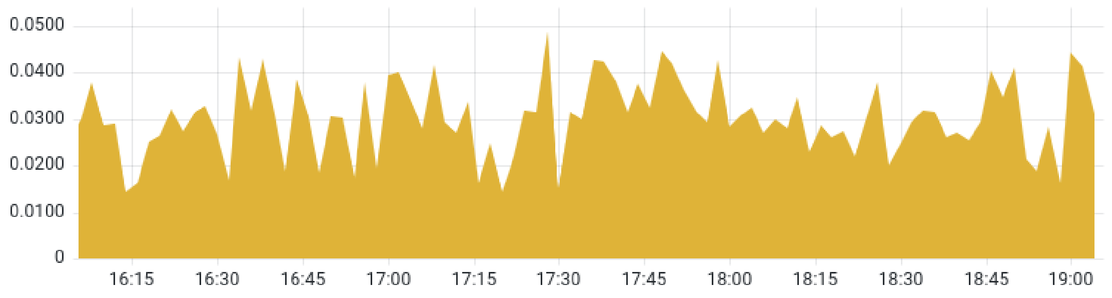


Data received (idle)

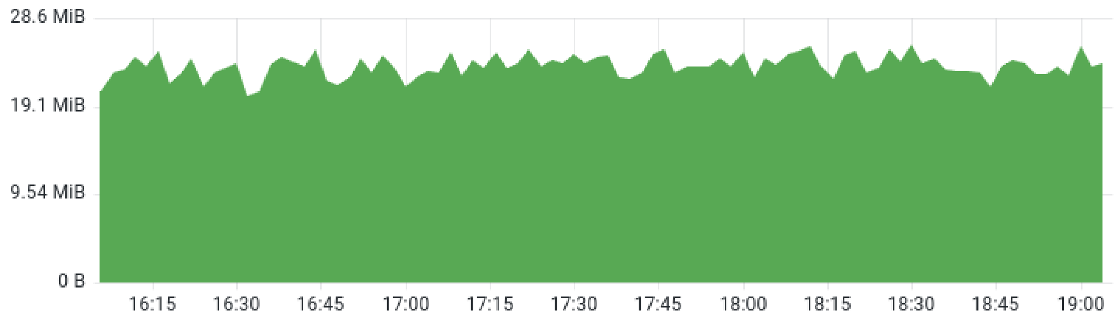


Data transmitted (idle)

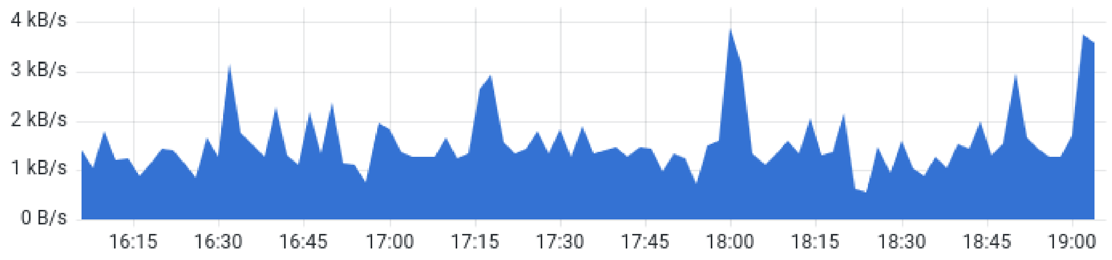
Figure 7.2: Resources consumed by liqo-gateway in Liqo 0.5.4



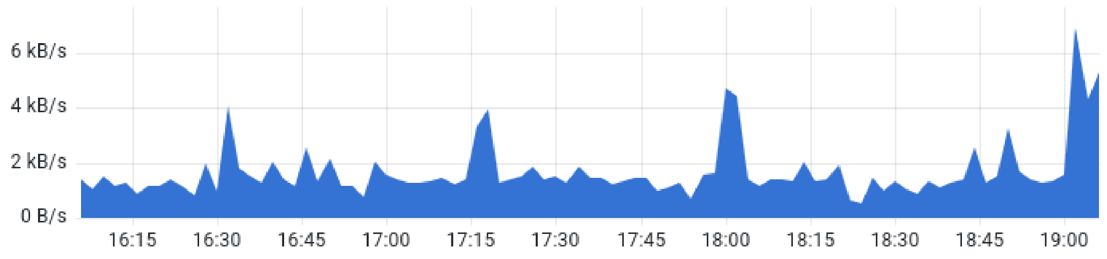
CPU consumption



Memory consumption



Data received (idle)



Data transmitted (idle)

Figure 7.3: Resources consumed by liqo-gateway in Liqo 0.6.0

Chapter 8

Conclusion

The necessity of multi-clusters approach has become a necessity in the industry. The possibility to allow different clusters to inter-operate, share resources and workloads will be a fundamental feature in the future of cloud computing. The thesis has a dual purpose, one was to analyze what could be the possible network architecture in a multi-clusters environment, and the other was to design and implement a way to monitor the network. In particular, the **cross-cluster** network part of Liqo has been studied. The current Liqo design has been compared with a new one, which proved to be more efficient and with fewer dependencies. It was a success and set a new direction for future development. Indeed (as discussed in chapter 7), Liqo has reached a certain level of maturity and the new design would be a breaking change. At the moment the current design works and the benefits would not justify the effort of a new one. The work for the thesis has remained a **proof of concept** which has given to the Liqo team a major awareness of what could be a direction to follow in the future, in case of a massive rework of the Liqo network.

However, this has not limited the design and implementation of a mechanism to monitor connection with other clusters. A monitoring system has been designed and integrated into Liqo and now the Liqo cross-cluster network status can be observed and monitored. In the near future, this system can be extended to the inner part of the Liqo network and could include also other components and parts of Liqo, to provide users the possibility to monitor Liqo entirely.

Bibliography

[1] *8 facts about real-world container use*. URL: https://www.datadoghq.com/container-report/ (cit. on p. 1).

[2] Joan Engebretson. *Will Kubernetes Be the Operating System for 5G? AT&T News Suggests Yes*. Feb. 2019. URL: https://www.telecompetitor.com/will-kubernetes-be-the-operating-system-for-5g-att-news-suggests-yes/ (cit. on p. 1).

[3] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. «Large-scale cluster management at Google with Borg». In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015 (cit. on p. 5).

[4] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. «Omega: flexible, scalable schedulers for large compute clusters». In: *SIGOPS European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, 2013, pp. 351–364. URL: http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf (cit. on p. 5).

[5] Ferenc Hámori. *The History of Kubernetes on a Timeline*. June 2018. URL: https://blog.risingstack.com/the-history-of-kubernetes/ (cit. on p. 5).

[6] Steven J. Vaughan-Nichols. *The five reasons Kubernetes won the container orchestration wars*. Jan. 2019. URL: https://blogs.dxc.technology/2019/01/28/the-five-reasons-kubernetes-won-the-container-orchestration-wars/ (cit. on p. 5).

[7] Kalyan Ramanathan. *5 business reasons why every CIO should consider Kubernetes*. Oct. 2019. URL: https://www.sumologic.com/blog/why-use-kubernetes/ (cit. on p. 5).

[8] *Kubernetes official documentation*. URL: https://kubernetes.io/docs/home/ (cit. on pp. 7, 12, 14–17, 21).

[9] Eric Carter. *Sysdig 2019 Container Usage Report: New Kubernetes and security insights.* Oct. 2019. URL: `https://sysdig.com/blog/sysdig-2019-container-usage-report/` (cit. on p. 8).

[10] *k8s Network Model.* URL: `https://kubernetes.io/docs/concepts/cluster-administration/networking/#the-kubernetes-network-model` (cit. on p. 17).

[11] *k8s CNI.* URL: `https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/` (cit. on p. 20).

[12] *k8s Services.* URL: `https://sookocheff.com/post/kubernetes/understanding-kubernetes-networking-model/` (cit. on p. 21).

[13] *Kubebuilder git repository.* URL: `https://github.com/kubernetes-sigs/kubebuilder` (cit. on p. 21).

[14] *Kubernetes Operator pattern.* URL: `https://kubernetes.io/docs/concepts/extend-kubernetes/operator/` (cit. on p. 21).

[15] *Liqo documentation.* URL: `https://docs.liqo.io/` (cit. on p. 22).

[16] *Liqo GitHub repository.* URL: `https://github.com/liqotech/liqo` (cit. on p. 22).

[17] *Virtual Kubelet GitHub repository.* URL: `https://github.com/virtual-kubelet/virtual-kubelet` (cit. on p. 27).